ATTACHMENT 1

```
  Filename: MP3Jukeboxx.java (a.k.a "Juke-A-Nator")

* Author: Tom Myers
*
* Version: 1.0
*
* Purpose: This file contains the entry point for the Juke-A-Nator application and is
*          responsible for:
*            1. Creating all the other main components of this application.
*            2. Displaying all of the user-interface components.
*            3. Performing all event-handling for the user-interface components.
*
* Inputs:
*         1. x - Where x is the number of credits to override any previous value.
*
* Outputs:
*        1. GbaMgr.DAT - A file containing the persistent state of the bill acceptor table.
*        2. MP3Jukeboxx.properties - A file containing configuration information for the app.
*        3. MP3Jukeboxx.PL - A file containing the outstanding playlist of songs to be added
*           to the song queue (a.k.a. playlist)
*        4. MP3Jukeboxx.LOG - A file containing log history for the application (max size 1MB)
*
* Development Environment:  JDK 1.3 (Sun's Java 2 Standard Edition version 1.3) was used to
* compile this java source file into MP3Jukeboxx.java.  Additionally, the javax.comm and
* Java Media Framework (JMF) libraries (both by Sun Microsystems) were used to compile various
* elements of this application.
*
*
* (c) Copyright 2000 Digital Jukebox Technologies LLC.  All Rights Reserved.
*/

import java.awt.*;
import java.awt.Insets;
import java.awt.event.*;
import java.util.Date;
import java.util.Vector;
import java.util.Enumeration;
import java.util.Properties;
import java.util.Random;
import java.io.*;
import java.net.URL;
import java.beans.PropertyChangeListener;

import javax.swing.*;
import javax.swing.tree.*;
import javax.swing.table.*;
import javax.swing.tree.DefaultTreeSelectionModel.*;
import javax.swing.ImageIcon;
import javax.swing.Icon;
import javax.swing.Timer;
import javax.swing.border.EmptyBorder;
import javax.swing.border.LineBorder;
import javax.swing.event.TreeSelectionListener;
import javax.swing.event.ListSelectionListener;
import javax.swing.event.ListSelectionEvent;
import javax.swing.event.ChangeListener;
import javax.swing.event.ChangeEvent;

import com.sun.java.swing.plaf.windows.WindowsLookAndFeel;

import TreeMgr.*;
import WinAmpMgr.*;
import GBAMgr.*;
import MyRenderer.*;
import MyListRenderer.*;
import AddPathDialog.*;
import LogonDialog.*;
import CDPanel.*;
import SpinButton.*;
import ConfirmationDialog.*;

public class MP3Jukeboxx extends JDialog implements ActionListener, WindowListener, MouseListener
{
   private boolean bDebug = false;
   private boolean bOwnerMode = false;


   private JPanel     cardPanel;
   private JPanel     userPanel;

   private CardLayout cardMgr;
   private CardLayout userCardMgr;

   private static final int CLASSIC_PANEL = 0;
   private static final int ADMIN_PANEL   = 1;
   private static final int TABLE_PANEL   = 2;
   private static final int SEARCH_PANEL  = 3;
   private static final int GENRE_PANEL   = 4;

```

```
91      private int iVisiblePanel = CLASSIC_PANEL;
92      private int iLastVisiblePanel = CLASSIC_PANEL;
93
94
95      // For the "classic" view.
96      // ----------------------------
97      private JPanel  classicPanel;
98      private CDPanel northwestCD;
99      private CDPanel northeastCD;
100     private CDPanel southwestCD;
101     private CDPanel southeastCD;
102
103
104     // For the "genre" view.
105     // ----------------------------
106     private JPanel      genrePanel;
107     private JLabel      genreLabel;
108     private Vector      genreTitleVect;
109     private JList       genreList;
110     private JScrollPane genreScrollPane;
111     private CDPanel     genreNorthCD;
112     private ImageIcon   genreNorthCDImage;
113     private Vector      genreNorthSongVector;
114     private String      genreNorthCDTitle;
115     private String      genreNorthGenre;
116     private CDPanel     genreSouthCD;
117     private ImageIcon   genreSouthCDImage;
118     private Vector      genreSouthSongVector;
119     private String      genreSouthCDTitle;
120     private String      genreSouthGenre;
121     private JButton     genrePageUpBtn;
122     private JButton     genrePageDnBtn;
123     private JButton     genreCloseBtn;
124
125
126     // For the "table" view.
127     //------------------------------
128     private JPanel            tablePanel;
129     private JScrollPane       tableScrollPane;
130     private DefaultTableModel tableModel;
131     private JTable            table;
132     private CDPanel           tableCDPanel;
133
134     private ImageIcon   viewImg;
135     private ImageIcon   rankingImg;
136     private ImageIcon   displayImg;
137
138     private JLabel viewLbl;
139     private JLabel rankingLbl;
140     private JLabel displayLbl;
141
142     private JButton tableSongViewBtn;
143     private JButton tableCDViewBtn;
144     private JButton tableAbsRankingBtn;
145     private JButton tablePwrRankingBtn;
146     private JButton tableShowTop50Btn;
147     private JButton tableShowTop100Btn;
148     private JButton tableShowAllBtn;
149     private JButton tableShowNewBtn;
150     private JButton tableCloseBtn;
151     private JButton tablePageUpBtn;
152     private JButton tablePageDnBtn;
153
154     private boolean bTableSongView   = true;
155     private boolean bTableAbsRanking = true;
156
157     private static final int TABLE_TOP50  = 1;
158     private static final int TABLE_TOP100 = 2;
159     private static final int TABLE_ALL    = 3;
160     private static final int TABLE_NEW    = 4;
161
162     private int     iTableSize = TABLE_TOP50;
163
164     private int         iRow       = -1;
165     private String      strSong    = null;
166     private String      strCDTitle = null;
167     private String      strGenre   = null;
168     private ImageIcon   coverImage = null;
169     private Vector      songVector = null;
170     private DefaultMutableTreeNode selectedNode = null;
171
172     private Vector            tableVector;
173
174     private Vector            top50Vect;
175     private Vector            top100Vect;
176     private Vector            allVect;
177
178     private Vector            top50VectByPwr;
179     private Vector            top100VectByPwr;
180     private Vector            allVectByPwr;
```

```
182     private Vector            top50CDVect;
183     private Vector            top100CDVect;
184     private Vector            allCDVect;
185     private Vector            newCDVect;
186
187     private Vector            top50CDVectByPwr;
188     private Vector            top100CDVectByPwr;
189     private Vector            allCDVectByPwr;
190     private Vector            newCDVectByPwr;
191
192     private Vector            columnHeaderVect;
193     private final String colRank  = "Rank";
194     private final String colAge   = "Age";
195     private final String colPlays = "Plays";
196     private final String colRatio = "Plays/Day";
197     private final String colGenre = "Genre";
198     private final String colCD    = "CD Title";
199     private final String colSong  = "Artist-Track-Song Name";
200     private final String colMp3   = "MP3 Object";
201
202     private boolean bPopularTableColumnsHidden = true;
203     private TableColumn tcAge         = null;
204     private TableColumn tcPlays       = null;
205     private TableColumn tcPlaysPerDay = null;
206
207
208     // For the "admin" view.
209     // ----------------------------
210     private JPanel            adminPanel;
211
212     private DefaultTreeModel  treeModel;
213     private JTree             tree;
214     private JScrollPane       treeScrollPane;
215
216     private JScrollPane       playlistScrollPane;
217     private JList             playlistList;
218     private Vector            playlistVector;
219     private JButton           adminNextBtn;
220     private JButton           adminPauseBtn;
221     private JButton           adminPlayBtn;
222     private JButton           adminMoveUpBtn;
223     private JButton           adminMoveDnBtn;
224     private JButton           adminRemoveBtn;
225
226     private JButton           ownerIncrementBtn;
227     private JButton           ownerDecrementBtn;
228     private JButton           ownerAddPathBtn;
229     private JButton           ownerDeleteFromDiskBtn;
230     private JButton           ownerAddNodeToQBtn;
231     private JButton           ownerRemNodeFromQBtn;
232     private JButton           ownerResetTreeBtn;
233     private JLabel            ownerNumToQueueLabel;
234     private SpinButton        ownerNumToQueueSB;
235     private JLabel            adminPlayerVolumeLabel;
236     private SpinButton        adminPlayerVolumeSB;
237     private JCheckBox         adminShowQueuedCB;
238     private JCheckBox         adminRandomPlayCB;
239     private JCheckBox         adminShowConfirmationCB;
240     private JLabel            adminRandomIntervalLabel;
241     private SpinButton        adminRandomIntervalSB;
242
243     private DefaultTableModel billStatsModel;
244     private JTable            billStats;
245     private JScrollPane       billStatsScrollPane;
246
247     private JTextArea         adminLogTextArea;
248     private JScrollPane       adminLogScrollPane;
249
250     private JLabel            treeLabel;
251     private JLabel            playlistLabel;
252     private JLabel            adminLogLabel;
253     private JLabel            billStatsLabel;
254
255
256
257     // For the "search" view.
258     // ----------------------------
259     private JPanel      searchPanel;
260     private boolean     state = false;
261     private int         iSearchBy = TreeMgr.BY_ALL;
262
263     private String      searchText   = null;
264     private JLabel      searchResult = null;
265     private JScrollPane searchScroll = null;
266     private DefaultTableModel searchTableModel = null;
267     private JTable      searchTable  = null;
268     private Vector      searchColumnHeaderVect = null;
269     private final String colCDNum = "CD #";
270     private Vector      searchVector = null;
271     private Vector      searchTableVector = null;
```

```
272       private JLabel      searchLabel  = null;
273
274       private JButton     searchSearch    = null;
275       private JButton     searchCancel    = null;
276       private JButton     searchClear     = null;
277       private JButton     searchPageUpBtn = null;
278       private JButton     searchPageDnBtn = null;
279
280       private JLabel      searchByLabel = null;
281
282       private JButton     searchByArtistBtn  = null;
283       private JButton     searchBySongBtn    = null;
284       private JButton     searchByCDTitleBtn = null;
285       private JButton     searchByAllBtn     = null;
286
287       private CDPanel     searchCDPanel    = null;
288       private String      searchStrSong    = null;
289       private String      searchStrCDTitle = null;
290       private String      searchStrGenre   = null;
291       private ImageIcon   searchCoverImage = null;
292       private Vector      searchSongVector = null;
293       private int         searchIndex      = 0;
294       private JTextField  searchTextField  = null;
295       private int         iSearchMp3Row    = -1;
296
297       private KeyboardPanel keyboardPanel = null;
298
299
300
301       // Bottom
302       private JPanel     bottomPanel;
303
304       private JButton treeViewBtn;
305       private JButton tableBtn;
306       private JButton genreBtn;
307       private JButton showCurrentBtn;
308       private JButton srchBtn;
309       private JButton topBtn;
310       private JButton prevGenreBtn;
311       private JButton nextGenreBtn;
312       private JButton btmBtn;
313
314       private JLabel     totalCDsLabel;
315       private JTextField totalCDsTxtField;
316
317       private JLabel     visibleCDsLabel;
318       private JTextField visibleCDsTxtField;
319
320       private JLabel     nowPlayingLabel;
321       private JTextField nowPlayingTxtField;
322       private JButton    prevPageBtn;
323       private JButton    nextPageBtn;
324
325       private JLabel     creditsLabel;
326       private JTextField creditsTxtField;
327       private JLabel     selectionLabel;
328       private JTextField selectionTxtField;
329       private JButton    btn_1;
330       private JButton    btn_2;
331       private JButton    btn_3;
332       private JButton    btn_4;
333       private JButton    btn_5;
334       private JButton    btn_6;
335       private JButton    btn_7;
336       private JButton    btn_8;
337       private JButton    btn_9;
338       private JButton    btn_0;
339       private JButton    cancelBtn;
340       private JButton    enterBtn;
341
342       // Data
343       private Timer  timer;
344       private Random random;
345
346       private int   credits;
347       private int   newCredits;
348       private Integer intCredits;
349       private String strText = "";
350
351       static final private int CURRENT  = 2;
352       static final private int PREVIOUS = 3;
353       static final private int NEXT     = 4;
354       static final private int TOP      = 5;
355       static final private int BTM      = 6;
356
357
358       private int      iMaxCDPtr = 0;
359       private int      iCurrentCDPtr = 0;
360       private Vector   CDVector;
361
```

```
362     private int      iCurrentGenrePtr = 0;
363     private int      iMaxGenrePtr = 0;
364     private Vector   GenreVector;
365     private Vector   GenreTitleVector;
366
367     public TreeMgr   treeMgr;
368     public PlayerMgr playerMgr;
369     public GBAMgr    gbaMgr;
370
371     private int      iLastSelTreeRow = 0;
372     private int      iElapsedRuntime = 1;         // when incremented by one, equals 500ms.
373     private int      iElapsedSilence = 1;         // when incremented by one, equals 500ms.
374     private int      iElapsedUserInactivity = 1;  // when incremented by one, equals 500ms.
375     private String   strCurrentSong;
376     private boolean  bChangedSong       = false;
377     private boolean  bDirtyFlag         = false;
378     private boolean  bIsAppFunctional   = false;
379     private boolean  bIsAppStarted      = false;
380     private boolean  bButtonsEnabled    = false;
381     private boolean  bEntryButtonsStale = true;
382
383
384     // The following are configuration parameters for the jukebox.
385     // These values are read from the properties file at initialization
386     // and can only be changed by editing this file directly.
387     // ---------------------------------------------------------------
388     private boolean bRandomPlay = true;
389     private Integer intRandomPlayInterval = new Integer(20);
390     private int     iRandomPlayInterval = 20;
391     // ---------------------------------------------------------------
392     private boolean bFlipToRandom = true;  // At startup, if set will flip to a random CD.
393     // ---------------------------------------------------------------
394     private boolean bShowQueued = true; // Whether or not the user sees the "queued" flag on songs.
395     // ---------------------------------------------------------------
396     private boolean bShowConfirmation = true; // Whether or not the user gets a confirmation query.
397     // ---------------------------------------------------------------
398     private Integer intNumberToQueue = new Integer(0); // If set to non-zero, that number of "free"
399     private int     iNumberToQueue = 0;                // songs will be maintained in the queue.
400     private Integer intPlayerVolume = new Integer(75);
401     private int     iPlayerVolume = 75;
402     // ---------------------------------------------------------------
403     private Integer intLevel_0 = new Integer(0);
404     private int     iLevel_0   = 0;
405
406     private Integer intLevel_1 = new Integer(5);
407     private int     iLevel_1   = 5;
408
409     private Integer intLevel_2 = new Integer(10);
410     private int     iLevel_2   = 10;
411     // ---------------------------------------------------------------
412     private int     iCreditsPer = 4; // The default is to award 4 credits for 1 $1.00 bill entered.
413     private Integer intCreditsPer = new Integer(4);
414
415     private int     iBonusLevel_1  = 8;   // If the user entered 2 $1.00 bills or 1 $2.00 bill
416     private Integer intBonusFactor_1 = new Integer(1);
417     private int     iBonusFactor_1 = 1;   // Give the user 1 bonus credit.
418
419     private int     iBonusLevel_2  = 20;  // If the user entered any combination of bill(s) totaling $5.00
420     private Integer intBonusFactor_2 = new Integer(3);
421     private int     iBonusFactor_2 = 3;   // Give the user 3 bonus credits.
422
423     private int     iBonusLevel_3  = 40;  // If the user entered any combination of bill(s) totaling $10.00
424     private Integer intBonusFactor_3 = new Integer(5);
425     private int     iBonusFactor_3 = 5;   // Give the user 5 bonus credits.
426
427     private int     iBonusLevel_4  = 80;  // If the user entered any combination of bill(s) totaling $20.00
428     private Integer intBonusFactor_4 = new Integer(7);
429     private int     iBonusFactor_4 = 7;   // Give the user 7 bonus credits.
430     // ---------------------------------------------------------------
431     private Integer intNewCDVectorSize = new Integer(50);
432     private int     iNewCDVectorSize = 50;
433
434     private Integer intNewCDAgeThreshold = new Integer(30); // In Days.
435     private int     iNewCDAgeThreshold = 30;
436     // ---------------------------------------------------------------
437
438
439     // For application settings persistence.
440     private Properties properties;
441
442
443     // For accounting purposes on the acceptor, this tracks the bills inserted.
444     private Vector acceptorVector;
445
446
447     // For logging.
448     static public BufferedWriter out = null;
449     static public File logFile = null;
450     static public RandomAccessFile raLogFile = null;
451     static public String strTrcHdr = "";
```

```
452     static public final int ENTER = 0;
453     static public final int EXIT = 1;
454     static public final int COMMENT = 2;
455
456
457     public MP3Jukeboxx(String strTitle, int initialCredits, boolean debug)
458     {
459        super();
460
461        trace("MP3Jukeboxx()", ENTER);
462
463        bDebug = debug;
464
465        getContentPane().setLayout(null);
466        setBounds(-3,-25,1030,800);
467        addWindowListener(this);
468
469
470        // Load the bill acceptor statistics file.
471        loadAcceptorVector();
472
473        // Load in the configuration file.
474        loadProperties();
475
476
477        // Pass in true so that it will create a numbering scheme for the CDs that it finds.
478        // (A CD will be considered a parent directory of songs).
479        treeMgr = new TreeMgr(true, true);
480
481
482        playerMgr = new WinAmpMgr();
483        playerMgr.setNumberToQueue(iNumberToQueue);
484        playerMgr.setVolume(iPlayerVolume);
485        playerMgr.setLockOnQueue(false);
486        playerMgr.releaseInitialLock();
487
488
489        gbaMgr = new GBAMgr();
490
491        timer  = new Timer(500, this);
492
493        random = new Random(java.lang.System.currentTimeMillis());
494
495
496        // Override any outstanding credits if the passed in credits parameter is non-zero.
497        if (initialCredits < 0)
498        {
499           intCredits = new Integer(0);
500           newCredits = 0;
501
502           // Force an update of the text field.
503           credits = 1;
504        }
505        else if (initialCredits > 0)
506        {
507           intCredits = new Integer(initialCredits);
508           newCredits = initialCredits;
509
510           // Force an update of the text field.
511           credits = 0;
512        }
513
514
515        initGui();
516        initTree();
517        initCDVector();
518        initGenreVector();
519        initGenreTitleVector();
520
521
522        classicPanel = new JPanel();
523        classicPanel.setBounds(0,0,1024,600);
524        classicPanel.setForeground(Color.white);
525        classicPanel.setBackground(Color.black);
526        classicPanel.setLayout(null);
527
528
529        if (bFlipToRandom == true)
530        {
531           flipToRandomCD();
532        }
533
534        initClassicPanel(CURRENT);
535        addToClassicPanel();
536
537        initTablePanel();
538        initGenrePanel();
539        initSearchPanel();
540
541        updateVisibleCDTextField();
```

```
542         selectionTxtField.setText("");
543
544         userPanel.add(classicPanel, "classic");
545         userPanel.add(tablePanel, "table");
546         userPanel.add(genrePanel, "genre");
547         userPanel.add(searchPanel, "search");
548
549
550         cardPanel.add(userPanel, "user");
551         cardPanel.add(adminPanel, "admin");
552
553
554         getContentPane().add(cardPanel);
555         getContentPane().add(bottomPanel);
556
557         setForeground(Color.white);
558         setBackground(Color.black);
559
560
561         // Add a "shutdown hook" function that will be called in case there is an unexpected termination
562         // signal to the Java VM.  In this case, all we want to do is save data to disk.
563         Runtime.getRuntime().addShutdownHook(
564            new Thread()
565            {
566               public void run()
567               {
568                  cleanup();
569               }
570            });
571
572
573         setResizable(false);
574         setVisible(true);
575
576         if (bIsAppFunctional == true && bIsAppStarted == false)
577         {
578            // Start the timer.
579            timer.start();
580
581            // Start playerMgr, which will do its thing in a separate thread.
582            playerMgr.start();
583         }
584
585
586         // The below bypasses security measures because it is assumed only authorized people will
587         // have access to the physical keyboard...
588         addKeyListener(
589            new KeyListener()
590            {
591               public void keyPressed(java.awt.event.KeyEvent event)
592               {
593                  char ch = event.getKeyChar();
594
595                  if (ch == java.awt.event.KeyEvent.VK_ESCAPE)
596                  {
597                     if (iVisiblePanel != ADMIN_PANEL)
598                     {
599                        logInfo("Owner mode.");
600
601                        bOwnerMode = true;
602                        visibleCDsTxtField.setText("");
603                        disableBottomPanel();
604
605                        ownerIncrementBtn.setVisible(true);
606                        ownerDecrementBtn.setVisible(true);
607                        ownerAddPathBtn.setVisible(true);
608                        ownerDeleteFromDiskBtn.setVisible(true);
609                        ownerAddNodeToQBtn.setVisible(true);
610                        ownerRemNodeFromQBtn.setVisible(true);
611                        ownerResetTreeBtn.setVisible(true);
612                        ownerNumToQueueSB.setVisible(true);
613
614                        iLastVisiblePanel = iVisiblePanel;
615                        iVisiblePanel = ADMIN_PANEL;
616
617                        loadLogFile();
618
619                        cardPanel.setVisible(false);
620
621                        cardMgr.show(cardPanel, "admin");
622                        cardPanel.setVisible(true);
623                     }
624                     else
625                     {
626                        logInfo("Showing User Panel.");
627                        bOwnerMode = false;
628
629                        iVisiblePanel = iLastVisiblePanel;
630                        iLastVisiblePanel = ADMIN_PANEL;
631                        checkBottomPanel();
```

```
                    cardPanel.setVisible(false);
                    cardMgr.show(cardPanel, "user");
                    cardPanel.setVisible(true);
                }
            }
        }
        public void keyTyped(java.awt.event.KeyEvent event) { }
        public void keyReleased(java.awt.event.KeyEvent event) { }
    });


    trace("MP3Jukeboxx()", EXIT);
}

private void flipToRandomCD()
{
    trace("flipToRandomCD()", ENTER);

    // First, generate a random number between 0 and 4 less than the Maximum number of CDs.
    int iRndCD = random.nextInt();
    iRndCD = Math.abs(iRndCD);
    iRndCD = iRndCD % (iMaxCDPtr - 4);
    iRndCD += 1;


    // Now, point to this random CD and make  it visible.
    iCurrentCDPtr = iRndCD;
    setSelectedCDForCDPtr(iCurrentCDPtr);


    // Get the corresponding Genre pointer.
    iCurrentGenrePtr = getGenrePtrForSelectedCD();


    // Now, enable/disable the scrolling buttons appropriately.
    checkScrollButtons();


    trace("***");
    trace("Random iCurrentCDPtr: " + iCurrentCDPtr);
    trace("***");

    trace("flipToRandomCD()", EXIT);
}

private void initSearchPanel()
{
    trace("initSearchPanel()", ENTER);

    searchPanel = new JPanel();
    searchPanel.setForeground(Color.white);
    searchPanel.setBackground(Color.black);
    searchPanel.setBounds(0,0,1024,600);
    searchPanel.setLayout(null);

    searchText   = null;
    searchVector = new Vector();
    searchIndex  = 0;

    searchLabel = new JLabel("Text To Search For:");
    searchLabel.setBounds(25,315,120,20);
    searchLabel.setFont(new Font("SansSerif", Font.BOLD, 12));
    searchLabel.setForeground(Color.yellow);
    searchPanel.add(searchLabel);

    searchTextField = new JTextField("");
    searchTextField.setBounds(140,315,360,20);
    searchTextField.setEditable(false);
    searchTextField.setFont(new Font("SansSerif", Font.BOLD, 12));
    searchTextField.setForeground(Color.black);
    searchTextField.setBackground(Color.white);
    searchPanel.add(searchTextField);

    searchSearch = new JButton();
    initControl(searchPanel, searchSearch, 10,255,90,50, true);
    searchSearch.setIcon(loadIcon("images/go.gif"));
    searchSearch.setDisabledIcon(loadIcon("images/godisabled.gif"));
    searchSearch.setPressedIcon(loadIcon("images/gopressed.gif"));
    searchSearch.setBorderPainted(false);
    searchSearch.setFocusPainted(false);

    searchClear  = new JButton();
    initControl(searchPanel, searchClear,  105,255,90,50, true);
    searchClear.setIcon(loadIcon("images/clear.gif"));
    searchClear.setDisabledIcon(loadIcon("images/cleardisabled.gif"));
    searchClear.setPressedIcon(loadIcon("images/clearpressed.gif"));
    searchClear.setBorderPainted(false);
    searchClear.setFocusPainted(false);

    searchCancel = new JButton();
```

```
723         initControl(searchPanel, searchCancel, 200,255,90,50, true);
724         searchCancel.setIcon(loadIcon("images/close.gif"));
725         searchCancel.setDisabledIcon(loadIcon("images/closedisabled.gif"));
726         searchCancel.setPressedIcon(loadIcon("images/closepressed.gif"));
727         searchCancel.setBorderPainted(false);
728         searchCancel.setFocusPainted(false);
729
730         searchPageUpBtn = new JButton();
731         searchPageUpBtn.setIcon(loadIcon("images/genrepageup.gif"));
732         searchPageUpBtn.setPressedIcon(loadIcon("images/genrepageuppressed.gif"));
733         searchPageUpBtn.setDisabledIcon(loadIcon("images/genrepageupdisabled.gif"));
734         searchPageUpBtn.setBorderPainted(false);
735         searchPageUpBtn.setFocusPainted(false);
736         searchPageUpBtn.setBounds(420,395,75,75);
737         searchPageUpBtn.setEnabled(true);
738         searchPageUpBtn.addActionListener(this);
739         searchPanel.add(searchPageUpBtn);
740
741         searchPageDnBtn = new JButton();
742         searchPageDnBtn.setIcon(loadIcon("images/genrepagedn.gif"));
743         searchPageDnBtn.setPressedIcon(loadIcon("images/genrepagednpressed.gif"));
744         searchPageDnBtn.setDisabledIcon(loadIcon("images/genrepagedndisabled.gif"));
745         searchPageDnBtn.setBorderPainted(false);
746         searchPageDnBtn.setFocusPainted(false);
747         searchPageDnBtn.setBounds(420,480,75,75);
748         searchPageDnBtn.setEnabled(true);
749         searchPageDnBtn.addActionListener(this);
750         searchPanel.add(searchPageDnBtn);
751
752         searchByLabel = new JLabel("Search By:");
753         searchByLabel.setBounds(727,253,175,30);
754         searchByLabel.setFont(new Font("SansSerif", Font.BOLD, 26));
755         searchByLabel.setForeground(Color.white);
756         searchPanel.add(searchByLabel);
757
758         searchByArtistBtn = new JButton();
759         searchByArtistBtn.setIcon(loadIcon("images/byartistenabled.gif"));
760         searchByArtistBtn.setPressedIcon(loadIcon("images/byartistpressed.gif"));
761         searchByArtistBtn.setDisabledIcon(loadIcon("images/byartistdisabled.gif"));
762         searchByArtistBtn.setBorderPainted(false);
763         searchByArtistBtn.setFocusPainted(false);
764         searchByArtistBtn.setBounds(860,247,85,22);
765         searchByArtistBtn.setEnabled(true);
766         searchByArtistBtn.addActionListener(this);
767         searchPanel.add(searchByArtistBtn);
768
769         searchBySongBtn = new JButton();
770         searchBySongBtn.setIcon(loadIcon("images/bysongenabled.gif"));
771         searchBySongBtn.setPressedIcon(loadIcon("images/bysongpressed.gif"));
772         searchBySongBtn.setDisabledIcon(loadIcon("images/bysongdisabled.gif"));
773         searchBySongBtn.setBorderPainted(false);
774         searchBySongBtn.setFocusPainted(false);
775         searchBySongBtn.setBounds(860,277,80,22);
776         searchBySongBtn.setEnabled(true);
777         searchBySongBtn.addActionListener(this);
778         searchPanel.add(searchBySongBtn);
779
780         searchByCDTitleBtn = new JButton();
781         searchByCDTitleBtn.setIcon(loadIcon("images/bytitleenabled.gif"));
782         searchByCDTitleBtn.setPressedIcon(loadIcon("images/bytitlepressed.gif"));
783         searchByCDTitleBtn.setDisabledIcon(loadIcon("images/bytitledisabled.gif"));
784         searchByCDTitleBtn.setBorderPainted(false);
785         searchByCDTitleBtn.setFocusPainted(false);
786         searchByCDTitleBtn.setBounds(950,247,73,22);
787         searchByCDTitleBtn.setEnabled(true);
788         searchByCDTitleBtn.addActionListener(this);
789         searchPanel.add(searchByCDTitleBtn);
790
791         searchByAllBtn = new JButton();
792         searchByAllBtn.setIcon(loadIcon("images/byallenabled.gif"));
793         searchByAllBtn.setPressedIcon(loadIcon("images/byallpressed.gif"));
794         searchByAllBtn.setDisabledIcon(loadIcon("images/byalldisabled.gif"));
795         searchByAllBtn.setBorderPainted(false);
796         searchByAllBtn.setFocusPainted(false);
797         searchByAllBtn.setBounds(950,277,57,22);
798         searchByAllBtn.setEnabled(false);
799         searchByAllBtn.addActionListener(this);
800         searchPanel.add(searchByAllBtn);
801
802
803         keyboardPanel = new KeyboardPanel(0,0,1024,300, searchTextField);
804         searchPanel.add(keyboardPanel);
805
806
807         searchColumnHeaderVect = new Vector();
808         searchColumnHeaderVect.addElement(colCDNum);
809         searchColumnHeaderVect.addElement(colSong);
810         searchColumnHeaderVect.addElement(colMp3);
811
812         searchTableVector = createSearchTableVector(searchVector);
```

```
813          searchTableModel = new DefaultTableModel(searchVector, searchColumnHeaderVect);
814          searchTable = createSearchTable(searchTableModel);
815
816          searchScroll = new JScrollPane();
817          searchScroll.setBounds(2,350,398,245);
818          searchScroll.setForeground(Color.white);
819          searchScroll.setBackground(Color.black);
820
821          searchScroll.getViewport().setForeground(Color.white);
822          searchScroll.getViewport().setBackground(Color.black);
823
824                   searchScroll.getViewport().add(searchTable);
825          searchScroll.setBorder(new LineBorder(Color.white, 2));
826
827          JScrollBar horizontal = searchScroll.getHorizontalScrollBar();
828          horizontal.setPreferredSize(new Dimension(horizontal.getWidth(),25));
829
830          JScrollBar vertical = searchScroll.getVerticalScrollBar();
831          vertical.setPreferredSize(new Dimension(25,vertical.getHeight()));
832          searchPanel.add(searchScroll);
833
834          trace("initSearchPanel", EXIT);
835       }
836
837       private void setSelectedCDForGenrePtr(int iGenrePtr)
838       {
839          trace("setSelectedCDForGenrePtr()", ENTER);
840
841          iSearchMp3Row = ((Integer)GenreVector.elementAt(iGenrePtr)).intValue();
842          tree.setSelectionRow(iSearchMp3Row);
843
844          trace("setSelectedCDForGenrePtr()", EXIT);
845       }
846
847       private void setSelectedCDForCDPtr(int iCDPtr)
848       {
849          trace("setSelectedCDForCDPtr()", ENTER);
850
851          iSearchMp3Row = ((Integer)CDVector.elementAt(iCDPtr)).intValue();
852          tree.setSelectionRow(iSearchMp3Row);
853
854          trace("setSelectedCDForCDPtr()", EXIT);
855       }
856
857       private int getGenrePtrForSelectedCD()
858       {
859          trace("getGenrePtrForSelectedCD()", ENTER);
860
861          int iGenrePtr = -1;
862          int iRow      = -1;
863          boolean bDone   = false;
864          Integer element = null;
865
866          int iSelRow = tree.getMaxSelectionRow();
867
868          if (iSelRow != -1)
869          {
870             for (Enumeration enum = GenreVector.elements(); enum.hasMoreElements() && !bDone; )
871             {
872                element = (Integer)enum.nextElement();
873
874                iRow = element.intValue();
875
876                if (iRow > iSelRow)
877                {
878                   bDone = true;
879                }
880                else
881                {
882                   iGenrePtr = GenreVector.indexOf(element);
883                }
884             }
885          }
886
887          trace("getGenrePtrForSelectedCD()", EXIT);
888
889          return iGenrePtr;
890       }
891
892       private int getCDPtrForSelectedCD()
893       {
894          trace("getCDPtrForSelectedCD()", ENTER);
895
896          int iCDPtr = -1;
897          int iRow   = -1;
898          boolean bDone   = false;
899          Integer element = null;
900
901          int iSelRow = tree.getMaxSelectionRow();
902
```

```
903          if (iSelRow != -1)
904          {
905             for (Enumeration enum = CDVector.elements(); enum.hasMoreElements() && !bDone; )
906             {
907                element = (Integer)enum.nextElement();
908
909                iRow = element.intValue();
910
911                if (iRow == iSelRow)
912                {
913                   bDone = true;
914
915                   iCDPtr = CDVector.indexOf(element);
916                }
917             }
918          }
919
920          trace("getCDPtrForSelectedCD()", EXIT);
921
922          return iCDPtr;
923       }
924
925       private void initSearchCDPanel()
926       {
927          trace("initSearchCDPanel()", ENTER);
928
929          searchPanel.setVisible(false);
930
931          if (searchCDPanel != null)
932          {
933             searchPanel.remove(searchCDPanel);
934             searchCDPanel.die();
935             searchCDPanel = null;
936          }
937
938          DefaultMutableTreeNode selectedNode = (DefaultMutableTreeNode)tree.getLastSelectedPathComponent();
939          searchCoverImage = treeMgr.getCoverImage(tree, selectedNode);
940
941          if (selectedNode.toString().length() >= 4)
942             searchStrCDTitle = selectedNode.toString().substring(0,4) + treeMgr.getCDArtist(selectedNode) + "-" +
selectedNode.toString().substring(4,selectedNode.toString().length());
943          else
944             searchStrCDTitle = selectedNode.toString();
945
946          searchSongVector = treeMgr.getCDAllChildren(selectedNode);
947          searchStrGenre = treeMgr.getGenre(selectedNode);
948
949          searchCDPanel = new CDPanel(iNewCDAgeThreshold, bShowQueued, iLevel_0, iLevel_1, iLevel_2, searchCoverImage
, searchSongVector, searchStrCDTitle, searchStrGenre, selectionTxtField, cancelBtn);
950          searchCDPanel.setBounds(515,301,509,295);
951
952          searchPanel.add(searchCDPanel);
953
954          visibleCDsTxtField.setText(searchCDPanel.getCDNumber());
955
956
957          searchCoverImage = null;
958          searchSongVector = null;
959          searchStrGenre   = null;
960          searchStrCDTitle = null;
961
962          searchPanel.setVisible(true);
963
964          trace("initSearchCDPanel()", EXIT);
965       }
966
967       private void initGenrePanel()
968       {
969          trace("initGenrePanel()", ENTER);
970
971          genrePanel = new JPanel();
972          genrePanel.setForeground(Color.white);
973          genrePanel.setBackground(Color.black);
974          genrePanel.setBounds(0,0,1024,600);
975          genrePanel.setLayout(null);
976
977          genreLabel = new JLabel("Please select a Genre:", JLabel.CENTER);
978          genreLabel.setBounds(25,20,325,20);
979          genreLabel.setForeground(Color.white);
980          genreLabel.setBackground(Color.black);
981          genreLabel.setFont(new Font("SansSerif", Font.BOLD, 18));
982          genrePanel.add(genreLabel);
983
984          genreTitleVect = new Vector();
985          for (Enumeration enum = GenreTitleVector.elements(); enum.hasMoreElements(); )
986          {
987             Vector rowVect = (Vector)enum.nextElement();
988             genreTitleVect.addElement((String)rowVect.elementAt(0));
989          }
990
```

```
991         genreList = new JList(genreTitleVect);
992         genreList.setForeground(Color.white);
993         genreList.setBackground(Color.black);
994         genreList.setBounds(0,0,325,475);
995         genreList.setCellRenderer(
996            new DefaultListCellRenderer()
997            {
998               public java.awt.Component getListCellRendererComponent(JList   list,
999                                                                       Object  value,
1000                                                                      int     index,
1001                                                                      boolean isSelected,
1002                                                                      boolean cellHasFocus)
1003
1004              {
1005                 super.getListCellRendererComponent(list,
1006                                                    value,
1007                                                    index,
1008                                                    isSelected,
1009                                                    cellHasFocus);
1010
1011                 setText("  " + getText());
1012                 setFont(new Font("SansSerif", Font.BOLD, 18));
1013                 setForeground(Color.yellow);
1014
1015                 return this;
1016              }
1017           });
1018
1019        genreList.addListSelectionListener(
1020           new ListSelectionListener()
1021           {
1022              public void valueChanged(javax.swing.event.ListSelectionEvent event)
1023              {
1024                 int i = genreList.getMaxSelectionIndex();
1025
1026                 if (i >= 0)
1027                 {
1028                    String strGenreKey = (String)genreTitleVect.elementAt(i);
1029                    int iGenreRow = 0;
1030
1031                    boolean bDone = false;
1032                    for (Enumeration enum = GenreTitleVector.elements(); enum.hasMoreElements() && !bDone; )
1033                    {
1034                       Vector rowVect = (Vector)enum.nextElement();
1035                       String strGenreTitle = (String)rowVect.elementAt(0);
1036
1037                       if (strGenreTitle.equals(strGenreKey))
1038                       {
1039                          bDone = true;
1040                          iGenreRow = ((Integer)rowVect.elementAt(1)).intValue();
1041                       }
1042                    }
1043
1044                    int iCDPtr = 0;
1045                    int iCDRow = ((Integer)CDVector.elementAt(iCDPtr)).intValue();
1046
1047                    while (iCDRow < iGenreRow)
1048                    {
1049                       iCDPtr = iCDPtr + 1;
1050                       iCDRow = ((Integer)CDVector.elementAt(iCDPtr)).intValue();
1051                    }
1052
1053                    // Now, select the first CD from the selected genre.
1054                    setSelectedCDForCDPtr(iCDPtr);
1055
1056                    addToGenrePanel(iCDPtr);
1057                 }
1058              }
1059           });
1060
1061        genreScrollPane = new JScrollPane();
1062        genreScrollPane.setBounds(25,50,325,475);
1063        genreScrollPane.getViewport().add(genreList);
1064
1065        JScrollBar horizontal = genreScrollPane.getHorizontalScrollBar();
1066        horizontal.setPreferredSize(new Dimension(horizontal.getWidth(),25));
1067
1068        JScrollBar vertical = genreScrollPane.getVerticalScrollBar();
1069        vertical.setPreferredSize(new Dimension(25,vertical.getHeight()));
1070
1071        genrePanel.add(genreScrollPane);
1072
1073        genreNorthCD = null;
1074        genreSouthCD = null;
1075
1076        genrePageUpBtn = new JButton();
1077        genrePageUpBtn.setIcon(loadIcon("images/genrepageup.gif"));
1078        genrePageUpBtn.setPressedIcon(loadIcon("images/genrepageuppressed.gif"));
1079        genrePageUpBtn.setBorderPainted(false);
1080        genrePageUpBtn.setFocusPainted(false);
1081        genrePageUpBtn.setBounds(420,215,75,75);
1082        genrePageUpBtn.setEnabled(true);
```

```
1081          genrePageUpBtn.addActionListener(this);
1082          genrePanel.add(genrePageUpBtn);
1083
1084          genrePageDnBtn = new JButton();
1085          genrePageDnBtn.setIcon(loadIcon("images/genrepagedn.gif"));
1086          genrePageDnBtn.setPressedIcon(loadIcon("images/genrepagednpressed.gif"));
1087          genrePageDnBtn.setBorderPainted(false);
1088          genrePageDnBtn.setFocusPainted(false);
1089          genrePageDnBtn.setBounds(420,310,75,75);
1090          genrePageDnBtn.setEnabled(true);
1091          genrePageDnBtn.addActionListener(this);
1092          genrePanel.add(genrePageDnBtn);
1093
1094          genreCloseBtn = new JButton();
1095          genreCloseBtn.setIcon(loadIcon("images/close.gif"));
1096          genreCloseBtn.setDisabledIcon(loadIcon("images/closedisabled.gif"));
1097          genreCloseBtn.setPressedIcon(loadIcon("images/closepressed.gif"));
1098          genreCloseBtn.setBorderPainted(false);
1099          genreCloseBtn.setFocusPainted(false);
1100          genreCloseBtn.setBounds(25,540,90,50);
1101          genreCloseBtn.setEnabled(true);
1102          genreCloseBtn.addActionListener(this);
1103          genrePanel.add(genreCloseBtn);
1104
1105          trace("initGenrePanel()", EXIT);
1106       }
1107
1108       private void addToGenrePanel(int iCDPtr)
1109       {
1110          trace("addToGenrePanel()", ENTER);
1111
1112          genrePanel.setVisible(false);
1113
1114
1115          // Make the corresponding selection for the main window.
1116          iCurrentCDPtr = iCDPtr;
1117
1118          initClassicPanel(CURRENT);
1119          addToClassicPanel();
1120
1121
1122          // Remove any CDPanels if they exist.
1123          if (genreNorthCD != null)
1124          {
1125             genrePanel.remove(genreNorthCD);
1126             genreNorthCD.die();
1127             genreNorthCD = null;
1128          }
1129
1130          if (genreSouthCD != null)
1131          {
1132             genrePanel.remove(genreSouthCD);
1133             genreSouthCD.die();
1134             genreSouthCD = null;
1135          }
1136
1137
1138          // Select the first CD from the selected genre.
1139          setSelectedCDForCDPtr(iCDPtr);
1140          iCurrentGenrePtr = getGenrePtrForSelectedCD();
1141
1142          DefaultMutableTreeNode selectedNode = (DefaultMutableTreeNode)tree.getLastSelectedPathComponent();
1143          genreNorthCDImage = treeMgr.getCoverImage(tree, selectedNode);
1144
1145          if (selectedNode.toString().length() >= 4)
1146             genreNorthCDTitle = selectedNode.toString().substring(0,4) + treeMgr.getCDArtist(selectedNode) + "-" +
selectedNode.toString().substring(4,selectedNode.toString().length());
1147          else
1148             genreNorthCDTitle = selectedNode.toString();
1149
1150          genreNorthSongVector = treeMgr.getCDAllChildren(selectedNode);
1151          genreNorthGenre = treeMgr.getGenre(selectedNode);
1152
1153          // Then, create the "north" CD Panel.
1154          genreNorthCD = new CDPanel(iNewCDAgeThreshold, bShowQueued, iLevel_0, iLevel_1, iLevel_2, genreNorthCDImage
, genreNorthSongVector, genreNorthCDTitle, genreNorthGenre, selectionTxtField, cancelBtn);
1155          genreNorthCD.setBounds(514,1,509,295);
1156          genrePanel.add(genreNorthCD);
1157
1158
1159          // Now, select the second CD from the selected genre (if possible).
1160          if (iCDPtr + 1 <= iMaxCDPtr)
1161          {
1162             setSelectedCDForCDPtr(iCDPtr + 1);
1163
1164             int iTmpGenrePtr = getGenrePtrForSelectedCD();
1165
1166             if (iTmpGenrePtr == iCurrentGenrePtr)
1167             {
1168                selectedNode = (DefaultMutableTreeNode)tree.getLastSelectedPathComponent();
```

```
1169                genreSouthCDImage = treeMgr.getCoverImage(tree, selectedNode);
1170
1171                if (selectedNode.toString().length() >= 4)
1172                   genreSouthCDTitle = selectedNode.toString().substring(0,4) + treeMgr.getCDArtist(selectedNode) + "
-" + selectedNode.toString().substring(4,selectedNode.toString().length());
1173                else
1174                   genreSouthCDTitle = selectedNode.toString();
1175
1176                genreSouthSongVector = treeMgr.getCDAllChildren(selectedNode);
1177                genreSouthGenre = treeMgr.getGenre(selectedNode);
1178
1179                // Then, create the "south" CD Panel.
1180                genreSouthCD = new CDPanel(iNewCDAgeThreshold, bShowQueued, iLevel_0, iLevel_1, iLevel_2,
genreSouthCDImage, genreSouthSongVector, genreSouthCDTitle, genreSouthGenre, selectionTxtField, cancelBtn);
1181                genreSouthCD.setBounds(514,300,509,295);
1182                genrePanel.add(genreSouthCD);
1183             }
1184          }
1185
1186
1187          // Update the "Visible CDs" text field to reflect the fact that only 2 CDs are being shown.
1188          String strFirst = genreNorthCD.getCDNumber();
1189          String strLast  = null;
1190          if (genreSouthCD != null)
1191          {
1192             strLast = genreSouthCD.getCDNumber();
1193             visibleCDsTxtField.setText(strFirst + " - " + strLast);
1194          }
1195          else
1196             visibleCDsTxtField.setText(strFirst);
1197
1198          genrePanel.setVisible(true);
1199
1200          trace("addToGenrePanel()", EXIT);
1201       }
1202
1203       private void initTablePanel()
1204       {
1205          trace("initTablePanel()", ENTER);
1206
1207          tablePanel = new JPanel();
1208          tablePanel.setForeground(Color.white);
1209          tablePanel.setBackground(Color.black);
1210          tablePanel.setBounds(0,0,1024,600);
1211          tablePanel.setLayout(null);
1212
1213          viewImg = new ImageIcon("images/view.gif");
1214          viewLbl = new JLabel(viewImg);
1215          viewLbl.setBounds(15,325,125,25);
1216          viewLbl.setForeground(Color.white);
1217          viewLbl.setBackground(Color.black);
1218          tablePanel.add(viewLbl);
1219
1220          rankingImg = new ImageIcon("images/ranking.gif");
1221          rankingLbl = new JLabel(rankingImg);
1222          rankingLbl.setBounds(140,325,125,25);
1223          rankingLbl.setForeground(Color.white);
1224          rankingLbl.setBackground(Color.black);
1225          tablePanel.add(rankingLbl);
1226
1227          displayImg = new ImageIcon("images/display.gif");
1228          displayLbl = new JLabel(displayImg);
1229          displayLbl.setBounds(280,325,125,25);
1230          displayLbl.setForeground(Color.white);
1231          displayLbl.setBackground(Color.black);
1232          tablePanel.add(displayLbl);
1233
1234          tableSongViewBtn = new JButton();
1235          tableSongViewBtn.setIcon(loadIcon("images/songview.gif"));
1236          tableSongViewBtn.setDisabledIcon(loadIcon("images/songviewdisabled.gif"));
1237          tableSongViewBtn.setPressedIcon(loadIcon("images/songviewpressed.gif"));
1238          tableSongViewBtn.setBorderPainted(false);
1239          tableSongViewBtn.setFocusPainted(false);
1240          tableSongViewBtn.setBounds(15,355,125,50);
1241          tableSongViewBtn.setEnabled(false);
1242          tableSongViewBtn.addActionListener(this);
1243          tablePanel.add(tableSongViewBtn);
1244
1245          tableCDViewBtn = new JButton();
1246          tableCDViewBtn.setIcon(loadIcon("images/cdview.gif"));
1247          tableCDViewBtn.setDisabledIcon(loadIcon("images/cdviewdisabled.gif"));
1248          tableCDViewBtn.setPressedIcon(loadIcon("images/cdviewpressed.gif"));
1249          tableCDViewBtn.setBorderPainted(false);
1250          tableCDViewBtn.setFocusPainted(false);
1251          tableCDViewBtn.setBounds(15,415,125,50);
1252          tableCDViewBtn.setEnabled(true);
1253          tableCDViewBtn.addActionListener(this);
1254          tablePanel.add(tableCDViewBtn);
1255
1256          tableAbsRankingBtn = new JButton();
```

```
1257        tableAbsRankingBtn.setIcon(loadIcon("images/normal.gif"));
1258        tableAbsRankingBtn.setDisabledIcon(loadIcon("images/normaldisabled.gif"));
1259        tableAbsRankingBtn.setPressedIcon(loadIcon("images/normalpressed.gif"));
1260        tableAbsRankingBtn.setBorderPainted(false);
1261        tableAbsRankingBtn.setFocusPainted(false);
1262        tableAbsRankingBtn.setBounds(140,355,125,50);
1263        tableAbsRankingBtn.setEnabled(false);
1264        tableAbsRankingBtn.addActionListener(this);
1265        tablePanel.add(tableAbsRankingBtn);
1266
1267        tablePwrRankingBtn = new JButton();
1268        tablePwrRankingBtn.setIcon(loadIcon("images/power.gif"));
1269        tablePwrRankingBtn.setDisabledIcon(loadIcon("images/powerdisabled.gif"));
1270        tablePwrRankingBtn.setPressedIcon(loadIcon("images/powerpressed.gif"));
1271        tablePwrRankingBtn.setBorderPainted(false);
1272        tablePwrRankingBtn.setFocusPainted(false);
1273        tablePwrRankingBtn.setBounds(140,415,125,50);
1274        tablePwrRankingBtn.setEnabled(true);
1275        tablePwrRankingBtn.addActionListener(this);
1276        tablePanel.add(tablePwrRankingBtn);
1277
1278        tableShowTop50Btn = new JButton();
1279        tableShowTop50Btn.setIcon(loadIcon("images/top50.gif"));
1280        tableShowTop50Btn.setDisabledIcon(loadIcon("images/top50disabled.gif"));
1281        tableShowTop50Btn.setPressedIcon(loadIcon("images/top50pressed.gif"));
1282        tableShowTop50Btn.setBorderPainted(false);
1283        tableShowTop50Btn.setFocusPainted(false);
1284        tableShowTop50Btn.setBounds(280,355,125,50);
1285        tableShowTop50Btn.setEnabled(false);
1286        tableShowTop50Btn.addActionListener(this);
1287        tablePanel.add(tableShowTop50Btn);
1288
1289        tableShowTop100Btn = new JButton();
1290        tableShowTop100Btn.setIcon(loadIcon("images/top100.gif"));
1291        tableShowTop100Btn.setDisabledIcon(loadIcon("images/top100disabled.gif"));
1292        tableShowTop100Btn.setPressedIcon(loadIcon("images/top100pressed.gif"));
1293        tableShowTop100Btn.setBorderPainted(false);
1294        tableShowTop100Btn.setFocusPainted(false);
1295        tableShowTop100Btn.setBounds(280,415,125,50);
1296        tableShowTop100Btn.setEnabled(true);
1297        tableShowTop100Btn.addActionListener(this);
1298        tablePanel.add(tableShowTop100Btn);
1299
1300        tableShowAllBtn = new JButton();
1301        tableShowAllBtn.setIcon(loadIcon("images/all.gif"));
1302        tableShowAllBtn.setDisabledIcon(loadIcon("images/alldisabled.gif"));
1303        tableShowAllBtn.setPressedIcon(loadIcon("images/allpressed.gif"));
1304        tableShowAllBtn.setBorderPainted(false);
1305        tableShowAllBtn.setFocusPainted(false);
1306        tableShowAllBtn.setBounds(280,475,125,50);
1307        tableShowAllBtn.setEnabled(true);
1308        tableShowAllBtn.addActionListener(this);
1309        tablePanel.add(tableShowAllBtn);
1310
1311        tableShowNewBtn = new JButton();
1312        tableShowNewBtn.setIcon(loadIcon("images/new.gif"));
1313        tableShowNewBtn.setDisabledIcon(loadIcon("images/newdisabled.gif"));
1314        tableShowNewBtn.setPressedIcon(loadIcon("images/newpressed.gif"));
1315        tableShowNewBtn.setBorderPainted(false);
1316        tableShowNewBtn.setFocusPainted(false);
1317        tableShowNewBtn.setBounds(280,535,125,50);
1318        tableShowNewBtn.setEnabled(true);
1319        tableShowNewBtn.setVisible(false);
1320        tableShowNewBtn.addActionListener(this);
1321        tablePanel.add(tableShowNewBtn);
1322
1323
1324        tablePageUpBtn = new JButton();
1325        tablePageUpBtn.setIcon(loadIcon("images/genrepageup.gif"));
1326        tablePageUpBtn.setPressedIcon(loadIcon("images/genrepageuppressed.gif"));
1327        tablePageUpBtn.setDisabledIcon(loadIcon("images/genrepageupdisabled.gif"));
1328        tablePageUpBtn.setBorderPainted(false);
1329        tablePageUpBtn.setFocusPainted(false);
1330        tablePageUpBtn.setBounds(420,395,75,75);
1331        tablePageUpBtn.setEnabled(true);
1332        tablePageUpBtn.addActionListener(this);
1333        tablePanel.add(tablePageUpBtn);
1334
1335        tablePageDnBtn = new JButton();
1336        tablePageDnBtn.setIcon(loadIcon("images/genrepagedn.gif"));
1337        tablePageDnBtn.setPressedIcon(loadIcon("images/genrepagednpressed.gif"));
1338        tablePageDnBtn.setDisabledIcon(loadIcon("images/genrepagedndisabled.gif"));
1339        tablePageDnBtn.setBorderPainted(false);
1340        tablePageDnBtn.setFocusPainted(false);
1341        tablePageDnBtn.setBounds(420,480,75,75);
1342        tablePageDnBtn.setEnabled(true);
1343        tablePageDnBtn.addActionListener(this);
1344        tablePanel.add(tablePageDnBtn);
1345
1346
```

```
1347         tableCloseBtn = new JButton();
1348         tableCloseBtn.setIcon(loadIcon("images/close.gif"));
1349         tableCloseBtn.setDisabledIcon(loadIcon("images/closedisabled.gif"));
1350         tableCloseBtn.setPressedIcon(loadIcon("images/closepressed.gif"));
1351         tableCloseBtn.setBorderPainted(false);
1352         tableCloseBtn.setFocusPainted(false);
1353         tableCloseBtn.setBounds(25,540,90,50);
1354         tableCloseBtn.setEnabled(true);
1355         tableCloseBtn.addActionListener(this);
1356         tablePanel.add(tableCloseBtn);
1357
1358
1359         columnHeaderVect = new Vector();
1360         columnHeaderVect.addElement(colRank);
1361         columnHeaderVect.addElement(colAge);
1362         columnHeaderVect.addElement(colPlays);
1363         columnHeaderVect.addElement(colRatio);
1364         columnHeaderVect.addElement(colGenre);
1365         columnHeaderVect.addElement(colCD);
1366         columnHeaderVect.addElement(colSong);
1367         columnHeaderVect.addElement(colMp3);
1368
1369         initTableVectors();
1370         tableVector = top50Vect; // The default view is top 50 songs, sorted by absolute rank.
1371
1372         tableModel = new DefaultTableModel(tableVector, columnHeaderVect);
1373         table = createTable(tableModel);
1374
1375         if (tableVector.size() > 0)
1376         {
1377            table.getSelectionModel().setLeadSelectionIndex(0);
1378            table.changeSelection(0, 1, false, false);
1379         }
1380         checkTableScrollButtons();
1381
1382         tableScrollPane = new JScrollPane();
1383         tableScrollPane.setBounds(2,2,1021,295);
1384         tableScrollPane.setBorder(new LineBorder(Color.white, 1));
1385         tableScrollPane.getViewport().setForeground(Color.white);
1386         tableScrollPane.getViewport().setBackground(Color.black);
1387                  tableScrollPane.getViewport().add(table);
1388
1389         JScrollBar horizontal = tableScrollPane.getHorizontalScrollBar();
1390         horizontal.setPreferredSize(new Dimension(horizontal.getWidth(),25));
1391
1392         JScrollBar vertical = tableScrollPane.getVerticalScrollBar();
1393         vertical.setPreferredSize(new Dimension(25,vertical.getHeight()));
1394
1395         tablePanel.add(tableScrollPane);
1396
1397         trace("initTablePanel()", EXIT);
1398      }
1399
1400      private void showTable()
1401      {
1402         trace("showTable()", ENTER);
1403
1404         switch (iTableSize)
1405         {
1406            case TABLE_NEW:
1407               if (!bTableSongView)
1408                  if (bTableAbsRanking)
1409                     tableVector = newCDVect;
1410                  else
1411                     tableVector = newCDVectByPwr;
1412               break;
1413
1414            case TABLE_ALL:
1415               if (bTableSongView)
1416                  if (bTableAbsRanking)
1417                     tableVector = allVect;
1418                  else
1419                     tableVector = allVectByPwr;
1420               else
1421                  if (bTableAbsRanking)
1422                     tableVector = allCDVect;
1423                  else
1424                     tableVector = allCDVectByPwr;
1425               break;
1426
1427            case TABLE_TOP100:
1428               if (bTableSongView)
1429                  if (bTableAbsRanking)
1430                     tableVector = top100Vect;
1431                  else
1432                     tableVector = top100VectByPwr;
1433               else
1434                  if (bTableAbsRanking)
1435                     tableVector = top100CDVect;
1436                  else
```

```
                tableVector = top100CDVectByPwr;
            break;

         case TABLE_TOP50:
            if (bTableSongView)
               if (bTableAbsRanking)
                  tableVector = top50Vect;
               else
                  tableVector = top50VectByPwr;
            else
               if (bTableAbsRanking)
                  tableVector = top50CDVect;
               else
                  tableVector = top50CDVectByPwr;
            break;
      }

      tableScrollPane.getViewport().remove(table);

      tableModel = new DefaultTableModel(tableVector, columnHeaderVect);
      table = createTable(tableModel);

      tableScrollPane.getViewport().add(table);


      if (tableVector.size() > 0)
      {
         table.getSelectionModel().setLeadSelectionIndex(0);
         table.changeSelection(0, 1, false, false);
      }

      selectionTxtField.setText("");

      trace("showTable()", EXIT);
   }

   private void initTableVectors()
   {
      trace("initTableVectors()", ENTER);

      // --------------------------------
      if (top50Vect == null)
         top50Vect = new Vector();
      else
         top50Vect.removeAllElements();
      // --------------------------------
      if (top100Vect == null)
         top100Vect = new Vector();
      else
         top100Vect.removeAllElements();
      // --------------------------------
      if (top50VectByPwr == null)
         top50VectByPwr = new Vector();
      else
         top50VectByPwr.removeAllElements();
      // --------------------------------
      if (top100VectByPwr == null)
         top100VectByPwr = new Vector();
      else
         top100VectByPwr.removeAllElements();
      // --------------------------------
      if (newCDVect == null)
         newCDVect = new Vector();
      else
         newCDVect.removeAllElements();
      // --------------------------------
      if (top50CDVect == null)
         top50CDVect = new Vector();
      else
         top50CDVect.removeAllElements();
      // --------------------------------
      if (top100CDVect == null)
         top100CDVect = new Vector();
      else
         top100CDVect.removeAllElements();
      // --------------------------------
      if (top50CDVectByPwr == null)
         top50CDVectByPwr = new Vector();
      else
         top50CDVectByPwr.removeAllElements();
      // --------------------------------
      if (top100CDVectByPwr == null)
         top100CDVectByPwr = new Vector();
      else
         top100CDVectByPwr.removeAllElements();
      // --------------------------------
      if (newCDVectByPwr == null)
         newCDVectByPwr = new Vector();
      else
         newCDVectByPwr.removeAllElements();
```

```
1528
1529        Vector rawAllVect        = treeMgr.getRankingVector(tree);
1530        Vector rawAllVectByPwr   = treeMgr.getPowerRankingVector(tree);
1531        Vector rawAllCDVect      = treeMgr.getCDRankingVector(tree);
1532        Vector rawAllCDVectByPwr = treeMgr.getCDPowerRankingVector(tree);
1533        Vector rawNewCDVect      = treeMgr.getNewCDRankingVector(tree, iNewCDAgeThreshold);
1534        Vector rawNewCDVectByPwr = treeMgr.getNewCDPowerRankingVector(tree, iNewCDAgeThreshold);
1535
1536        allVect = null;
1537        allVectByPwr = null;
1538        allCDVect = null;
1539        allCDVectByPwr = null;
1540
1541        allVect = new Vector();
1542        allVectByPwr = new Vector();
1543        allCDVect = new Vector();
1544        allCDVectByPwr = new Vector();
1545
1546
1547
1548        // Fill up the "All" Vector.
1549        PlayListEntry mp3 = null;
1550        int iRank = 0;
1551        int iNumPlaysMax = -1;
1552        int iNumPlays = 0;
1553        for (Enumeration enum = rawAllVect.elements(); enum.hasMoreElements(); )
1554        {
1555           mp3 = (PlayListEntry)enum.nextElement();
1556
1557           if (bDebug)
1558              System.out.println("Adding to <allVect>: " + mp3.toString());
1559
1560           iNumPlays = mp3.getPaidCnt();
1561
1562           if (iNumPlays != iNumPlaysMax)
1563           {
1564              iRank = iRank + 1;
1565              iNumPlaysMax = iNumPlays;
1566           }
1567
1568           addTableEntry(allVect, iRank, mp3);
1569        }
1570
1571
1572
1573        // Fill up the "AllByPwr" Vector.
1574        int iAge = 1;
1575        double dblMaxRatio = -1;
1576        double dblRatio = 1;
1577        iRank = 0;
1578        for (Enumeration enum = rawAllVectByPwr.elements(); enum.hasMoreElements(); )
1579        {
1580           mp3 = (PlayListEntry)enum.nextElement();
1581
1582           if (bDebug)
1583              System.out.println("Adding to <allVectByPwr>: " + mp3.toString());
1584
1585           iNumPlays = mp3.getPaidCnt();
1586           iAge = mp3.getAge();
1587
1588           if (iAge == 0)
1589              iAge = 1;
1590
1591           dblRatio = ((double)iNumPlays) / ((double)iAge);
1592
1593           if (dblRatio != dblMaxRatio)
1594           {
1595              iRank = iRank + 1;
1596              dblMaxRatio = dblRatio;
1597           }
1598
1599           addTableEntry(allVectByPwr, iRank, mp3);
1600        }
1601
1602
1603
1604        // Fill up the "AllCD" Vector.
1605        DefaultMutableTreeNode node = null;
1606        DefaultMutableTreeNode childNode = null;
1607        iRank = 0;
1608        int iSum = 0;
1609        int iMaxSum = -1;
1610        for (Enumeration enum = rawAllCDVect.elements(); enum.hasMoreElements(); )
1611        {
1612           node = (DefaultMutableTreeNode)enum.nextElement();
1613
1614           if (bDebug)
1615              System.out.println("Adding to <allCDVect>: " + node.toString());
1616
1617           iSum = 0;
```

```
1618            try
1619            {
1620               for (Enumeration enumCD = node.children(); enumCD.hasMoreElements(); )
1621               {
1622                  childNode = (DefaultMutableTreeNode)enumCD.nextElement();
1623
1624                  if (treeMgr.isPlayListEntry(childNode))
1625                  {
1626                     mp3 = (PlayListEntry)childNode.getUserObject();
1627                     iSum = iSum + mp3.getPaidCnt();
1628                  }
1629               }
1630            }
1631            catch (java.lang.ArrayIndexOutOfBoundsException idxExc)
1632            {
1633               System.out.println("No children for: " + childNode.toString());
1634            }
1635
1636            if (iSum != iMaxSum)
1637            {
1638               iRank = iRank + 1;
1639               iMaxSum = iSum;
1640            }
1641
1642            addTableEntry(allCDVect, iRank, node);
1643         }
1644
1645
1646
1647         // Fill up the "AllCDByPwr" Vector.
1648         iRank = 0;
1649         dblMaxRatio = -1;
1650         for (Enumeration enum = rawAllCDVectByPwr.elements(); enum.hasMoreElements(); )
1651         {
1652            node = (DefaultMutableTreeNode)enum.nextElement();
1653
1654            if (bDebug)
1655               System.out.println("Adding to <allCDVectByPwr>: " + node.toString());
1656
1657            iSum = 0;
1658            try
1659            {
1660               for (Enumeration enumCD = node.children(); enumCD.hasMoreElements(); )
1661               {
1662                  childNode = (DefaultMutableTreeNode)enumCD.nextElement();
1663
1664                  if (treeMgr.isPlayListEntry(childNode))
1665                  {
1666                     mp3 = (PlayListEntry)childNode.getUserObject();
1667                     iSum = iSum + mp3.getPaidCnt();
1668                     iAge = mp3.getAge();
1669                  }
1670               }
1671            }
1672            catch (java.lang.ArrayIndexOutOfBoundsException idxExc)
1673            {
1674               System.out.println("No children for: " + childNode.toString());
1675            }
1676
1677            if (iAge == 0)
1678               iAge = 1;
1679
1680            dblRatio = ((double)iSum) / ((double)iAge);
1681
1682            if (dblRatio != dblMaxRatio)
1683            {
1684               iRank = iRank + 1;
1685               dblMaxRatio = dblRatio;
1686            }
1687
1688            addTableEntry(allCDVectByPwr, iRank, node);
1689         }
1690
1691         // Fill up the "New CDVect" Vector.
1692         int iVectIndex = 0;
1693         iRank = 0;
1694         int iMaxAge = -1;
1695         boolean bDone = false;
1696         Vector rowVect = null;
1697         for (Enumeration enum = rawNewCDVect.elements(); enum.hasMoreElements() && !bDone; )
1698         {
1699            if (iVectIndex < iNewCDVectorSize)
1700            {
1701               iVectIndex = iVectIndex + 1;
1702
1703               node = (DefaultMutableTreeNode)enum.nextElement();
1704
1705               if (bDebug)
1706                  System.out.println("Adding to <newCDVect>: " + node.toString());
1707
```

```
1708            childNode = (DefaultMutableTreeNode)node.getFirstChild();
1709            if (treeMgr.isPlayListEntry(childNode))
1710            {
1711               mp3 = (PlayListEntry)childNode.getUserObject();
1712               iAge = mp3.getAge();
1713            }
1714
1715            if (iAge != iMaxAge)
1716            {
1717               iRank = iRank + 1;
1718               iMaxAge = iAge;
1719            }
1720
1721            addTableEntry(newCDVect, iRank, node);
1722         }
1723         else
1724         {
1725            bDone = true;
1726         }
1727      }
1728
1729
1730      // Fill up the "New CDVectByPwr" Vector.
1731      iVectIndex = 0;
1732      iRank = 0;
1733      dblMaxRatio = -1;
1734      bDone = false;
1735      rowVect = null;
1736      for (Enumeration enum = rawNewCDVectByPwr.elements(); enum.hasMoreElements() && !bDone; )
1737      {
1738         if (iVectIndex < iNewCDVectorSize)
1739         {
1740            iVectIndex = iVectIndex + 1;
1741
1742            node = (DefaultMutableTreeNode)enum.nextElement();
1743
1744            if (bDebug)
1745               System.out.println("Adding to <newCDVectByPwr>: " + node.toString());
1746
1747            iSum = 0;
1748            try
1749            {
1750               for (Enumeration enumCD = node.children(); enumCD.hasMoreElements(); )
1751               {
1752                  childNode = (DefaultMutableTreeNode)enumCD.nextElement();
1753
1754                  if (treeMgr.isPlayListEntry(childNode))
1755                  {
1756                     mp3 = (PlayListEntry)childNode.getUserObject();
1757                     iSum = iSum + mp3.getPaidCnt();
1758                     iAge = mp3.getAge();
1759                  }
1760               }
1761            }
1762            catch (java.lang.ArrayIndexOutOfBoundsException idxExc)
1763            {
1764               System.out.println("No children for: " + childNode.toString());
1765            }
1766
1767            if (iAge == 0)
1768               iAge = 1;
1769
1770            dblRatio = ((double)iSum) / ((double)iAge);
1771
1772            if (dblRatio != dblMaxRatio)
1773            {
1774               iRank = iRank + 1;
1775               dblMaxRatio = dblRatio;
1776            }
1777
1778            addTableEntry(newCDVectByPwr, iRank, node);
1779         }
1780         else
1781         {
1782            bDone = true;
1783         }
1784      }
1785
1786
1787
1788      // Fill up the "Top 50" Vector.
1789      iVectIndex = 0;
1790      bDone = false;
1791      for (Enumeration enum = allVect.elements(); enum.hasMoreElements() && !bDone; )
1792      {
1793         if (iVectIndex < 50)
1794         {
1795            if (bDebug)
1796               System.out.println("Adding element to <top50Vect> at index: " + iVectIndex);
1797
```

```
1798                top50Vect.addElement((Vector)enum.nextElement());
1799
1800                iVectIndex = iVectIndex + 1;
1801             }
1802             else
1803             {
1804                bDone = true;
1805             }
1806          }
1807
1808
1809          // Fill up the "Top 50ByPwr" Vector.
1810          iVectIndex = 0;
1811          bDone = false;
1812          for (Enumeration enum = allVectByPwr.elements(); enum.hasMoreElements() && !bDone; )
1813          {
1814             if (iVectIndex < 50)
1815             {
1816                if (bDebug)
1817                   System.out.println("Adding element to <top50VectByPwr> at index: " + iVectIndex);
1818
1819                top50VectByPwr.addElement((Vector)enum.nextElement());
1820
1821                iVectIndex = iVectIndex + 1;
1822             }
1823             else
1824             {
1825                bDone = true;
1826             }
1827          }
1828
1829
1830          // Fill up the "Top 50CD" Vector.
1831          iVectIndex = 0;
1832          bDone = false;
1833          for (Enumeration enum = allCDVect.elements(); enum.hasMoreElements() && !bDone; )
1834          {
1835             if (iVectIndex < 50)
1836             {
1837                if (bDebug)
1838                   System.out.println("Adding element to <top50CDVect> at index: " + iVectIndex);
1839
1840                top50CDVect.addElement((Vector)enum.nextElement());
1841
1842                iVectIndex = iVectIndex + 1;
1843             }
1844             else
1845             {
1846                bDone = true;
1847             }
1848          }
1849
1850
1851          // Fill up the "Top 50CDByPwr" Vector.
1852          iVectIndex = 0;
1853          bDone = false;
1854          for (Enumeration enum = allCDVectByPwr.elements(); enum.hasMoreElements() && !bDone; )
1855          {
1856             if (iVectIndex < 50)
1857             {
1858                if (bDebug)
1859                   System.out.println("Adding element to <top50CDVectByPwr> at index: " + iVectIndex);
1860
1861                top50CDVectByPwr.addElement((Vector)enum.nextElement());
1862
1863                iVectIndex = iVectIndex + 1;
1864             }
1865             else
1866             {
1867                bDone = true;
1868             }
1869          }
1870
1871
1872          // Fill up the "Top 100" Vector.
1873          iVectIndex = 0;
1874          bDone = false;
1875          for (Enumeration enum = allVect.elements(); enum.hasMoreElements() && !bDone; )
1876          {
1877             if (iVectIndex < 100)
1878             {
1879                if (bDebug)
1880                   System.out.println("Adding element to <top100Vect> at index: " + iVectIndex);
1881
1882                top100Vect.addElement((Vector)enum.nextElement());
1883
1884                iVectIndex = iVectIndex + 1;
1885             }
1886             else
1887             {
```

```
            bDone = true;
         }
      }


      // Fill up the "Top 100ByPwr" Vector.
      iVectIndex = 0;
      bDone = false;
      for (Enumeration enum = allVectByPwr.elements(); enum.hasMoreElements() && !bDone; )
      {
         if (iVectIndex < 100)
         {
            if (bDebug)
               System.out.println("Adding element to <top100VectByPwr> at index: " + iVectIndex);

            top100VectByPwr.addElement((Vector)enum.nextElement());

            iVectIndex = iVectIndex + 1;
         }
         else
         {
            bDone = true;
         }
      }


      // Fill up the "Top 100CD" Vector.
      iVectIndex = 0;
      bDone = false;
      for (Enumeration enum = allCDVect.elements(); enum.hasMoreElements() && !bDone; )
      {
         if (iVectIndex < 100)
         {
            if (bDebug)
               System.out.println("Adding element to <top100CDVect> at index: " + iVectIndex);

            top100CDVect.addElement((Vector)enum.nextElement());

            iVectIndex = iVectIndex + 1;
         }
         else
         {
            bDone = true;
         }
      }


      // Fill up the "Top 100CDByPwr" Vector.
      iVectIndex = 0;
      bDone = false;
      for (Enumeration enum = allCDVectByPwr.elements(); enum.hasMoreElements() && !bDone; )
      {
         if (iVectIndex < 100)
         {
            if (bDebug)
               System.out.println("Adding element to <top100CDVectByPwr> at index: " + iVectIndex);

            top100CDVectByPwr.addElement((Vector)enum.nextElement());

            iVectIndex = iVectIndex + 1;
         }
         else
         {
            bDone = true;
         }
      }

      trace("initTableVectors()", EXIT);
   }

   private void addTableEntry(Vector vect, int index, PlayListEntry mp3)
   {
      trace("addTableEntry()", ENTER);

      Integer intRank     = new Integer(index);
      Integer intAge      = new Integer(mp3.getAge());
      Integer intNumPlays = new Integer(mp3.getPaidCnt());

      java.text.NumberFormat nf = java.text.NumberFormat.getInstance(java.util.Locale.US);
      nf.setMaximumFractionDigits(2);
      String strNumPlaysPerDay = null;

      if (intAge.intValue() > 0)
         strNumPlaysPerDay = nf.format( intNumPlays.doubleValue() / intAge.doubleValue() );
      else
         strNumPlaysPerDay = intNumPlays.toString();

      String pathname = mp3.getMp3Path();
      int iFirstSlashIndex = pathname.indexOf("\\");
      int iNextSlashIndex = pathname.indexOf("\\", iFirstSlashIndex + 1);
```

```
1978        String strGenre = null;
1979
1980        if (iFirstSlashIndex != -1 && iNextSlashIndex != -1)
1981           strGenre = pathname.substring(iFirstSlashIndex+1, iNextSlashIndex);
1982        else
1983           strGenre = " ";
1984
1985
1986        String strCDTitle = null;
1987        if ( (strGenre.toLowerCase().indexOf("soundtrack") == -1) && iFirstSlashIndex != -1 && iNextSlashIndex != -
1)
1988        {
1989           iFirstSlashIndex = pathname.indexOf("\\", iNextSlashIndex + 1);
1990           iNextSlashIndex = pathname.indexOf("\\", iFirstSlashIndex + 1);
1991
1992        }
1993        else
1994        {
1995           iFirstSlashIndex = iNextSlashIndex;
1996           iNextSlashIndex = pathname.indexOf("\\", iFirstSlashIndex + 1);
1997        }
1998
1999        if (iFirstSlashIndex != -1 && iNextSlashIndex != -1)
2000           strCDTitle = pathname.substring(iFirstSlashIndex+1, iNextSlashIndex);
2001        else
2002           strCDTitle = " ";
2003
2004        if (strCDTitle == null)
2005           strCDTitle = " ";
2006
2007
2008        String strArtistTrackSong = mp3.toString().substring(0, mp3.toString().length()-4);
2009
2010        Vector rowVect = new Vector();
2011
2012        rowVect.addElement(intRank);
2013        rowVect.addElement(intAge);
2014        rowVect.addElement(intNumPlays);
2015        rowVect.addElement(strNumPlaysPerDay);
2016        rowVect.addElement(strGenre);
2017        rowVect.addElement(strCDTitle);
2018        rowVect.addElement(strArtistTrackSong);
2019
2020        // Hidden Column.
2021        rowVect.addElement(mp3);
2022
2023        vect.addElement(rowVect);
2024
2025        trace("addTableEntry()", EXIT);
2026     }
2027
2028     private void addTableEntry(Vector vect, int index, DefaultMutableTreeNode node)
2029     {
2030        trace("addTableEntry()", ENTER);
2031
2032        DefaultMutableTreeNode childNode = null;
2033        DefaultMutableTreeNode parentNode = null;
2034        DefaultMutableTreeNode grandParentNode = null;
2035
2036        PlayListEntry mp3 = null;
2037        Integer intRank   = new Integer(index);
2038        Integer intAge    = null;
2039        int iSumOfPlays   = 0;
2040
2041        for (Enumeration enumCD = node.children(); enumCD.hasMoreElements(); )
2042        {
2043           childNode = (DefaultMutableTreeNode)enumCD.nextElement();
2044
2045           if (treeMgr.isPlayListEntry(childNode))
2046           {
2047              mp3 = (PlayListEntry)childNode.getUserObject();
2048
2049              if (intAge == null)
2050              {
2051                 intAge = new Integer(mp3.getAge());
2052              }
2053
2054              iSumOfPlays = iSumOfPlays + mp3.getPaidCnt();
2055           }
2056        }
2057
2058        if (intAge == null)
2059        {
2060           intAge = new Integer(0);
2061        }
2062
2063        Integer intNumPlays = new Integer(iSumOfPlays);
2064
2065        java.text.NumberFormat nf = java.text.NumberFormat.getInstance(java.util.Locale.US);
2066        nf.setMaximumFractionDigits(2);
```

```
2067          String strNumPlaysPerDay = null;
2068
2069          if (intAge.intValue() > 0)
2070             strNumPlaysPerDay = nf.format( intNumPlays.doubleValue() / intAge.doubleValue() );
2071          else
2072             strNumPlaysPerDay = intNumPlays.toString();
2073
2074
2075          String strArtistTrackSong = null;
2076          String strGenre = null;
2077
2078          parentNode = (DefaultMutableTreeNode)node.getParent();
2079          if (parentNode.toString().toLowerCase().indexOf("soundtrack") >= 0)
2080          {
2081             strGenre = parentNode.toString();
2082             strArtistTrackSong = "Compilation";
2083          }
2084          else
2085          {
2086             grandParentNode = (DefaultMutableTreeNode)parentNode.getParent();
2087             strGenre = grandParentNode.toString();
2088             strArtistTrackSong = parentNode.toString();
2089
2090             if (strArtistTrackSong.toLowerCase().indexOf("compilation") >= 0)
2091             {
2092                strArtistTrackSong = "Compilation";
2093             }
2094          }
2095
2096          String strCDTitle = node.toString();
2097
2098
2099          Vector rowVect = new Vector();
2100
2101          rowVect.addElement(intRank);
2102          rowVect.addElement(intAge);
2103          rowVect.addElement(intNumPlays);
2104          rowVect.addElement(strNumPlaysPerDay);
2105          rowVect.addElement(strGenre);
2106          rowVect.addElement(strCDTitle);
2107          rowVect.addElement(strArtistTrackSong);
2108          rowVect.addElement(mp3);
2109
2110          vect.addElement(rowVect);
2111
2112          trace("addTableEntry()", EXIT);
2113       }
2114
2115       private Vector createSearchTableVector(Vector rawVector)
2116       {
2117          trace("createSearchTableVector()", ENTER);
2118
2119          Vector vect = new Vector();
2120          PlayListEntry mp3 = null;
2121
2122          if (rawVector.size() > 0)
2123          {
2124             for (Enumeration enum = rawVector.elements(); enum.hasMoreElements(); )
2125             {
2126                mp3 = (PlayListEntry)enum.nextElement();
2127                String strCDNum = treeMgr.getCDNumberForSong(tree, mp3);
2128                String strArtistTrackSong = mp3.toString().substring(0, mp3.toString().length()-4);
2129
2130                Vector rowVect = new Vector();
2131
2132                rowVect.addElement(strCDNum);
2133                rowVect.addElement(strArtistTrackSong);
2134                rowVect.addElement(mp3);
2135
2136                vect.addElement(rowVect);
2137             }
2138          }
2139          else
2140          {
2141             Vector rowVect = new Vector();
2142
2143             rowVect.addElement(" ");
2144             rowVect.addElement(" ");
2145
2146             // Hidden Column.
2147             rowVect.addElement(" ");
2148
2149             vect.addElement(rowVect);
2150          }
2151
2152          trace("createSearchTableVector()", EXIT);
2153
2154          return vect;
2155       }
2156
```

```
2157      private JTable createTable(DefaultTableModel model)
2158      {
2159         trace("createTable()", ENTER);
2160
2161         table = new JTable(model);
2162
2163         table.setBounds(0,0,510,594);
2164         table.setForeground(Color.white);
2165         table.setBackground(Color.black);
2166         table.setBorder(new LineBorder(Color.white, 1));
2167         table.setColumnSelectionAllowed(false);
2168         table.setCellSelectionEnabled(false);
2169         table.setRowSelectionAllowed(true);
2170         table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
2171         table.getSelectionModel().addListSelectionListener(
2172            new ListSelectionListener()
2173            {
2174               public void valueChanged(ListSelectionEvent e)
2175               {
2176                  PlayListEntry mp3 = null;
2177                  int i = table.getSelectionModel().getMaxSelectionIndex();
2178
2179                  // First, bring up the CD Panel for the selected song.
2180                  if (i >= 0 && i < tableModel.getDataVector().size())
2181                  {
2182                     mp3 = (PlayListEntry)tableModel.getValueAt(i, 7);
2183
2184                     if (mp3.getMp3Path() != null)
2185                     {
2186                        // We are dealing with a "Song View" table model.
2187                        if (treeMgr.setSelectedCDParentRowBySong(tree, mp3))
2188                        {
2189                           tablePanel.setVisible(false);
2190
2191                           if (tableCDPanel != null)
2192                           {
2193                              tablePanel.remove(tableCDPanel);
2194                              tableCDPanel.die();
2195                              tableCDPanel = null;
2196                           }
2197
2198                           DefaultMutableTreeNode selectedNode = (DefaultMutableTreeNode)tree.
getLastSelectedPathComponent();
2199                           coverImage = treeMgr.getCoverImage(tree, selectedNode);
2200
2201                           if (selectedNode.toString().length() >= 4)
2202                              strCDTitle = selectedNode.toString().substring(0,4) + treeMgr.getCDArtist(selectedNode
) + "-" + selectedNode.toString().substring(4,selectedNode.toString().length());
2203                           else
2204                              strCDTitle = selectedNode.toString();
2205
2206                           songVector = treeMgr.getCDAllChildren(selectedNode);
2207                           strGenre = treeMgr.getGenre(selectedNode);
2208
2209                           tableCDPanel = new CDPanel(iNewCDAgeThreshold, bShowQueued, iLevel_0, iLevel_1, iLevel_2,
coverImage, songVector, strCDTitle, strGenre, selectionTxtField, cancelBtn);
2210                           tableCDPanel.setBounds(514,301,509,295);
2211
2212                           tablePanel.add(tableCDPanel);
2213
2214                           // Update the "Visible CDs" text field to reflect the fact that only 1 CD is visible here
.
2215                           visibleCDsTxtField.setText(tableCDPanel.getCDNumber());
2216
2217                           tablePanel.setVisible(true);
2218                        }
2219                     }
2220                     else
2221                     {
2222                        // We are dealing with the "CD View" table model.
2223
2224                     }
2225                  }
2226
2227                  // Then, make the song the selection in its CD Panel (thus making the selection).
2228                  if (tableCDPanel != null && mp3 != null)
2229                  {
2230                     tableCDPanel.setSelectedSong(mp3);
2231                  }
2232
2233                  // Set the availability of the scroll buttons.
2234                  checkTableScrollButtons();
2235
2236                  /*
2237                  // Make the corresponding selection for the main window.
2238                  iCurrentCDPtr = getCDPtrForSelectedCD();
2239                  iCurrentGenrePtr = getGenrePtrForSelectedCD();
2240
2241                  initClassicPanel(CURRENT);
2242                  addToClassicPanel();
```

```
2243                checkScrollButtons();
2244                */
2245             }
2246          });
2247
2248
2249       table.setRowHeight(25);
2250       table.getColumn((Object)colRank).setPreferredWidth(30);
2251       table.getColumn((Object)colAge).setPreferredWidth(30);
2252       table.getColumn((Object)colPlays).setPreferredWidth(30);
2253       table.getColumn((Object)colRatio).setPreferredWidth(50);
2254       table.getColumn((Object)colGenre).setPreferredWidth(150);
2255       table.getColumn((Object)colCD).setPreferredWidth(250);
2256       table.getColumn((Object)colSong).setPreferredWidth(350);
2257
2258       table.getColumnModel().removeColumn(table.getColumn("MP3 Object"));
2259
2260       tcAge         = table.getColumn("Age");
2261       tcPlays       = table.getColumn("Plays");
2262       tcPlaysPerDay = table.getColumn("Plays/Day");
2263
2264       table.getColumnModel().removeColumn(tcAge);
2265       table.getColumnModel().removeColumn(tcPlays);
2266       table.getColumnModel().removeColumn(tcPlaysPerDay);
2267
2268       trace("createTable()", EXIT);
2269
2270       return table;
2271    }
2272
2273    private JTable createBillStatsTable()
2274    {
2275       trace("createBillStatsTable()", ENTER);
2276
2277       boolean bIsEmpty = true;
2278
2279       Vector colHdr = new Vector();
2280       colHdr.addElement("Month/Year");
2281       colHdr.addElement("Amount");
2282
2283       Vector dataVector = new Vector();
2284       for (Enumeration enum = acceptorVector.elements(); enum.hasMoreElements(); )
2285       {
2286          Vector rowVect = new Vector();
2287
2288          Vector keyValueVect = (Vector)enum.nextElement();
2289
2290          String  strMonthAndYear = (String)keyValueVect.elementAt(0);
2291          Integer amt = (Integer)keyValueVect.elementAt(1);
2292
2293
2294          String  strAmt = "$" + amt.toString() + ".00";
2295
2296
2297          rowVect.addElement(strMonthAndYear);
2298          rowVect.addElement(strAmt);
2299
2300          dataVector.addElement(rowVect);
2301
2302          if (bIsEmpty == true)
2303             bIsEmpty = false;
2304       }
2305
2306       if (bIsEmpty == true)
2307       {
2308          Vector rowVect = new Vector();
2309
2310          rowVect.addElement(" ");
2311          rowVect.addElement(" ");
2312
2313          dataVector.addElement(rowVect);
2314       }
2315
2316       billStatsModel = new DefaultTableModel(dataVector, colHdr);
2317
2318       billStats = new JTable(billStatsModel);
2319
2320       billStats.setBounds(0,0,451,275);
2321       billStats.setForeground(Color.white);
2322       billStats.setBackground(Color.black);
2323       billStats.setBorder(new LineBorder(Color.white, 1));
2324       billStats.setColumnSelectionAllowed(false);
2325       billStats.setCellSelectionEnabled(false);
2326       billStats.setRowSelectionAllowed(true);
2327       billStats.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
2328
2329       trace("createBillStatsTable()", EXIT);
2330
2331       return billStats;
2332    }
```

```
2334      private JTable createSearchTable(DefaultTableModel model)
2335      {
2336         trace("createSearchTable()", ENTER);
2337
2338         searchTable = new JTable(model);
2339
2340         searchTable.setBounds(0,0,400,370);
2341         searchTable.setForeground(Color.white);
2342         searchTable.setBackground(Color.black);
2343         searchTable.setBorder(new LineBorder(Color.white, 1));
2344         searchTable.setColumnSelectionAllowed(false);
2345         searchTable.setCellSelectionEnabled(false);
2346         searchTable.setRowSelectionAllowed(true);
2347         searchTable.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
2348         searchTable.getSelectionModel().addListSelectionListener(
2349            new ListSelectionListener()
2350            {
2351               public void valueChanged(ListSelectionEvent e)
2352               {
2353                  PlayListEntry mp3 = null;
2354                  int i = searchTable.getSelectionModel().getMaxSelectionIndex();
2355
2356                  // First, bring up the CD Panel for the selected song.
2357                  if (i >= 0 && i < searchTableModel.getDataVector().size())
2358                  {
2359                     mp3 = (PlayListEntry)searchTableModel.getValueAt(i, 2);
2360
2361                     if (treeMgr.setSelectedCDParentRowBySong(tree, mp3))
2362                     {
2363                        /*
2364                        // Make the corresponding selection for the main window.
2365                        iCurrentCDPtr = getCDPtrForSelectedCD();
2366                        iCurrentGenrePtr = getGenrePtrForSelectedCD();
2367
2368                        initClassicPanel(CURRENT);
2369                        addToClassicPanel();
2370                        checkScrollButtons();
2371                        */
2372
2373                        // Flag the marker so that if the user wants to go to the next CD, page, they can.
2374                        iSearchMp3Row = tree.getMaxSelectionRow();
2375
2376                        // Based on the selected song in the tree, show the appropriate CD Panel.
2377                        initSearchCDPanel();
2378
2379                        // Now, make the song the selection in its CD Panel (thus making the selection).
2380                        if (searchCDPanel != null && mp3 != null)
2381                        {
2382                           searchCDPanel.setSelectedSong(mp3);
2383                        }
2384                     }
2385                  }
2386
2387                  // Set the availability of the scroll buttons.
2388                  checkSearchScrollButtons();
2389               }
2390            });
2391
2392         searchTable.setRowHeight(22);
2393         searchTable.getColumn((Object)colCDNum).setPreferredWidth(40);
2394         searchTable.getColumn((Object)colSong).setPreferredWidth(360);
2395
2396         searchTable.getColumnModel().removeColumn(searchTable.getColumn("MP3 Object"));
2397
2398         trace("createSearchTable()", EXIT);
2399
2400         return searchTable;
2401      }
2402
2403      private int initClassicPanel(int iAction)
2404      {
2405         trace("initClassicPanel()", ENTER);
2406
2407         int       iOldCDPtr = iCurrentCDPtr;
2408         int       iCnt = 0;
2409
2410         int       iRow;
2411         String    strCDTitle;
2412         String    strGenre;
2413         ImageIcon coverImage;
2414         Vector    songVector;
2415         DefaultMutableTreeNode selectedNode;
2416
2417         classicPanel.removeAll();
2418
2419         if (northwestCD != null)
2420         {
2421            northwestCD.die();
2422            northwestCD = null;
2423         }
```

```
2425        if (northeastCD != null)
2426        {
2427           northeastCD.die();
2428           northeastCD = null;
2429        }
2430
2431        if (southwestCD != null)
2432        {
2433           southwestCD.die();
2434           southwestCD = null;
2435        }
2436
2437        if (southeastCD != null)
2438        {
2439           southeastCD.die();
2440           southeastCD = null;
2441        }
2442
2443
2444        switch (iAction)
2445        {
2446           case CURRENT:
2447
2448              break;
2449
2450           case NEXT:
2451              if (iCurrentCDPtr + 4 <= iMaxCDPtr)
2452              {
2453                 iCurrentCDPtr = iCurrentCDPtr + 4;
2454                 iCnt = 4;
2455              }
2456              else if (iCurrentCDPtr + 3 <= iMaxCDPtr)
2457              {
2458                 iCurrentCDPtr = iCurrentCDPtr + 3;
2459                 iCnt = 3;
2460              }
2461              else if (iCurrentCDPtr + 2 <= iMaxCDPtr)
2462              {
2463                 iCurrentCDPtr = iCurrentCDPtr + 2;
2464                 iCnt = 2;
2465              }
2466              else if (iCurrentCDPtr + 1 <= iMaxCDPtr)
2467              {
2468                 iCurrentCDPtr = iCurrentCDPtr + 1;
2469                 iCnt = 1;
2470              }
2471              else
2472              {
2473                 iCurrentCDPtr = iMaxCDPtr;
2474                 iCnt = 0;
2475              }
2476              break;
2477
2478           case PREVIOUS:
2479              if (iCurrentCDPtr - 4 >= 0)
2480              {
2481                 iCurrentCDPtr = iCurrentCDPtr - 4;
2482                 iCnt = 4;
2483              }
2484              else if (iCurrentCDPtr - 3 >= 0)
2485              {
2486                 iCurrentCDPtr = iCurrentCDPtr - 3;
2487                 iCnt = 3;
2488              }
2489              else if (iCurrentCDPtr - 2 >= 0)
2490              {
2491                 iCurrentCDPtr = iCurrentCDPtr - 2;
2492                 iCnt = 2;
2493              }
2494              else if (iCurrentCDPtr - 1 >= 0)
2495              {
2496                 iCurrentCDPtr = iCurrentCDPtr - 1;
2497                 iCnt = 1;
2498              }
2499              else
2500              {
2501                 iCurrentCDPtr = 0;
2502                 iCnt = 0;
2503              }
2504              break;
2505
2506           default:
2507              break;
2508        }
2509
2510
2511        if (iCurrentCDPtr <= iMaxCDPtr && iCurrentCDPtr >= 0)
2512        {
2513           iRow = ((Integer)CDVector.elementAt(iCurrentCDPtr)).intValue();
2514           tree.setSelectionRow(iRow);
```

```
2516            int iMaxRow = ((Integer)CDVector.elementAt(iMaxCDPtr)).intValue();
2517            tree.scrollRowToVisible(iMaxRow);
2518            tree.scrollRowToVisible(iRow);
2519
2520            selectedNode = (DefaultMutableTreeNode)tree.getLastSelectedPathComponent();
2521            coverImage = treeMgr.getCoverImage(tree, selectedNode);
2522
2523            if (selectedNode.toString().length() >= 4)
2524               strCDTitle = selectedNode.toString().substring(0,4) + treeMgr.getCDArtist(selectedNode) + "-" +
selectedNode.toString().substring(4,selectedNode.toString().length());
2525            else
2526               strCDTitle = selectedNode.toString();
2527
2528            songVector = treeMgr.getCDAllChildren(selectedNode);
2529            strGenre = treeMgr.getGenre(selectedNode);
2530
2531            northwestCD = new CDPanel(iNewCDAgeThreshold, bShowQueued, iLevel_0, iLevel_1, iLevel_2, coverImage,
songVector, strCDTitle, strGenre, selectionTxtField, cancelBtn);
2532            northwestCD.setBounds(1,1,509,295);
2533
2534            iCurrentCDPtr = iCurrentCDPtr + 1;
2535         }
2536
2537
2538         if (iCurrentCDPtr <= iMaxCDPtr && iCurrentCDPtr >= 0)
2539         {
2540            iRow = ((Integer)CDVector.elementAt(iCurrentCDPtr)).intValue();
2541            tree.setSelectionRow(iRow);
2542            selectedNode = (DefaultMutableTreeNode)tree.getLastSelectedPathComponent();
2543            coverImage = treeMgr.getCoverImage(tree, selectedNode);
2544
2545            if (selectedNode.toString().length() >= 4)
2546               strCDTitle = selectedNode.toString().substring(0,4) + treeMgr.getCDArtist(selectedNode) + "-" +
selectedNode.toString().substring(4,selectedNode.toString().length());
2547            else
2548               strCDTitle = selectedNode.toString();
2549
2550            songVector = treeMgr.getCDAllChildren(selectedNode);
2551            strGenre = treeMgr.getGenre(selectedNode);
2552
2553            southwestCD = new CDPanel(iNewCDAgeThreshold, bShowQueued, iLevel_0, iLevel_1, iLevel_2, coverImage,
songVector, strCDTitle, strGenre, selectionTxtField, cancelBtn);
2554            southwestCD.setBounds(1,300,509,295);
2555
2556            iCurrentCDPtr = iCurrentCDPtr + 1;
2557         }
2558
2559
2560         if (iCurrentCDPtr <= iMaxCDPtr && iCurrentCDPtr >= 0)
2561         {
2562            iRow = ((Integer)CDVector.elementAt(iCurrentCDPtr)).intValue();
2563            tree.setSelectionRow(iRow);
2564            selectedNode = (DefaultMutableTreeNode)tree.getLastSelectedPathComponent();
2565            coverImage = treeMgr.getCoverImage(tree, selectedNode);
2566
2567            if (selectedNode.toString().length() >= 4)
2568               strCDTitle = selectedNode.toString().substring(0,4) + treeMgr.getCDArtist(selectedNode) + "-" +
selectedNode.toString().substring(4,selectedNode.toString().length());
2569            else
2570               strCDTitle = selectedNode.toString();
2571
2572            songVector = treeMgr.getCDAllChildren(selectedNode);
2573            strGenre = treeMgr.getGenre(selectedNode);
2574
2575            northeastCD = new CDPanel(iNewCDAgeThreshold, bShowQueued, iLevel_0, iLevel_1, iLevel_2, coverImage,
songVector, strCDTitle, strGenre, selectionTxtField, cancelBtn);
2576            northeastCD.setBounds(514,1,509,295);
2577
2578            iCurrentCDPtr = iCurrentCDPtr + 1;
2579         }
2580
2581
2582         if (iCurrentCDPtr <= iMaxCDPtr && iCurrentCDPtr >= 0)
2583         {
2584            iRow = ((Integer)CDVector.elementAt(iCurrentCDPtr)).intValue();
2585            tree.setSelectionRow(iRow);
2586            selectedNode = (DefaultMutableTreeNode)tree.getLastSelectedPathComponent();
2587            coverImage = treeMgr.getCoverImage(tree, selectedNode);
2588
2589            if (selectedNode.toString().length() >= 4)
2590               strCDTitle = selectedNode.toString().substring(0,4) + treeMgr.getCDArtist(selectedNode) + "-" +
selectedNode.toString().substring(4,selectedNode.toString().length());
2591            else
2592               strCDTitle = selectedNode.toString();
2593
2594            songVector = treeMgr.getCDAllChildren(selectedNode);
2595            strGenre = treeMgr.getGenre(selectedNode);
2596
2597            southeastCD = new CDPanel(iNewCDAgeThreshold, bShowQueued, iLevel_0, iLevel_1, iLevel_2, coverImage,
songVector, strCDTitle, strGenre, selectionTxtField, cancelBtn);
```

```
2598            southeastCD.setBounds(514,300,509,295);
2599         }
2600
2601         iCurrentCDPtr = iOldCDPtr;
2602         iRow = ((Integer)CDVector.elementAt(iCurrentCDPtr)).intValue();
2603         tree.setSelectionRow(iRow);
2604
2605
2606         strCDTitle = null;
2607         strGenre   = null;
2608         coverImage = null;
2609         songVector = null;
2610
2611
2612         trace("initClassicPanel()", EXIT);
2613         return iCnt;
2614      }
2615
2616
2617      /**
2618       * Peforms a preorder traversal of the Jukebox tree.  For each "CD Node", that is,
2619       * for each node that starts with some alphanumeric sequence "xxx-CD Title", where
2620       * "xxx" corresponds to the CD number and "-" is a separator, and the rest is the
2621       * title of the CD, --- for each CD Node, there will be an entry in a one dimensional
2622       * arrary, called iCDArray, where the value will be the row index for the CD Node.
2623       *
2624       * For example, for the following tree structure:
2625       * <pre>
2626       * index   Node
2627       * -------------------
2628       * 0       Root
2629       * 1          D:
2630       * 2             Rock
2631       * 3                INXS
2632       * 4                   001-The Greatest Hits
2633       * .                   .
2634       * .                   .
2635       * .                   .
2636       * 16                  002-Kick
2637       * .                   .
2638       * .                   .
2639       * .                   .
2640       * 27                U2
2641       * 28                  003-The Joshua Tree
2642       * .                   .
2643       * .                   .
2644       * .                   .
2645       * 40             Industrial
2646       * 41                Rob Zombie
2647       * 42                  004-Hellbilly Deluxe
2648       * .                   .
2649       * .                   .
2650       * .                   .
2651       * </pre>
2652       *
2653       * The following array would be created:
2654       * <pre>
2655       * index  Value
2656       * -------------
2657       *   0      4
2658       *   1     16
2659       *   2     28
2660       *   3     42
2661       *   .      .
2662       *   .      .
2663       *   .      .
2664       * </pre>
2665       *
2666       *  This array would be used to determine how to load the 3 CDPanel panels (each panel would
2667       *  contain 4 CDPanels).  At program startup, the current panel would display CDs 0 to 3,
2668       *  the previous panel would be null (the previous button would be disabled),  and the next
2669       *  panel would be loaded in a separate thread to contain CDs 4 to 7.
2670       */
2671      private void initCDVector()
2672      {
2673         trace("initCDVector()", ENTER);
2674
2675         CDVector = treeMgr.getAllCDNodeChildren(tree);
2676
2677         iCurrentCDPtr = 0;
2678         iMaxCDPtr = CDVector.size() - 1;
2679
2680         Integer intTotalCDs = new Integer(CDVector.size());
2681         treeMgr.setNewCDIndex(intTotalCDs.intValue());
2682         totalCDsTxtField.setText(intTotalCDs.toString());
2683
2684         visibleCDsTxtField.setText("000 - 003");
2685
2686         logInfo("iCurrentCDPtr: " + iCurrentCDPtr);
2687         logInfo("iMaxCDPtr:     " + iMaxCDPtr);
```

```
2688         logInfo(" ");
2689
2690         trace("initCDVector()", EXIT);
2691      }
2692
2693      private void initGenreVector()
2694      {
2695         trace("initGenreVector()", ENTER);
2696
2697         GenreVector = treeMgr.getAllGenreNodeChildren(tree);
2698
2699         iCurrentGenrePtr = 0;
2700         iMaxGenrePtr = GenreVector.size() - 1;
2701
2702         logInfo("iCurrentGenrePtr: " + iCurrentGenrePtr);
2703         logInfo("iMaxGenrePtr:     " + iMaxGenrePtr);
2704         logInfo(" ");
2705
2706         trace("initCDVector()", EXIT);
2707      }
2708
2709      private void initGenreTitleVector()
2710      {
2711         trace("initGenreTitleVector()", ENTER);
2712
2713         GenreTitleVector = new Vector();
2714
2715         for (Enumeration enum = GenreVector.elements(); enum.hasMoreElements(); )
2716         {
2717            // Select the row for the next genre.
2718            Integer intGenreRow = (Integer)enum.nextElement();
2719            tree.setSelectionRow(intGenreRow.intValue());
2720
2721            // Get its title.
2722            Object selectedNode  = tree.getLastSelectedPathComponent();
2723            String strGenreTitle = selectedNode.toString();
2724
2725            // Now, add the title and row as a vector to the GenreTitleVector.
2726            Vector rowVect = new Vector();
2727
2728            if (bDebug)
2729               System.out.println("Adding to GenreTitleVector: " + strGenreTitle + ":" + intGenreRow.toString());
2730
2731            rowVect.addElement(strGenreTitle);
2732            rowVect.addElement(intGenreRow);
2733
2734            GenreTitleVector.addElement(rowVect);
2735         }
2736
2737         trace("initGenreTitleVector()", EXIT);
2738      }
2739
2740      private void loadAcceptorVector()
2741      {
2742         trace("loadAcceptorVector()", ENTER);
2743
2744         try
2745         {
2746            FileInputStream     in  = new FileInputStream("GBAMgr.DAT");
2747            BufferedInputStream buf = new BufferedInputStream(in);
2748            ObjectInputStream   s   = new ObjectInputStream(buf);
2749
2750            try
2751            {
2752               acceptorVector = (Vector)s.readObject();
2753            }
2754            catch (java.lang.ClassNotFoundException e2)
2755            {
2756              acceptorVector = null;
2757            }
2758            in.close();
2759            s.close();
2760
2761            in  = null;
2762            buf = null;
2763            s   = null;
2764         }
2765         catch (java.io.IOException e)
2766         {
2767            acceptorVector = null;
2768         }
2769
2770         if (acceptorVector == null)
2771         {
2772            acceptorVector = new Vector();
2773         }
2774
2775         // Now, create today's entry (initially zero).
2776         logStackedBill(0);
2777
```

```
2778        trace("loadAcceptorVector()", EXIT);
2779     }
2780
2781     private void saveAcceptorVector()
2782     {
2783        trace("saveAcceptorVector()", ENTER);
2784
2785        try
2786        {
2787           FileOutputStream     out = new FileOutputStream("GBAMgr.DAT");
2788           BufferedOutputStream buf = new BufferedOutputStream(out);
2789           ObjectOutputStream     s = new ObjectOutputStream(buf);
2790
2791           s.writeObject(acceptorVector);
2792
2793           s.flush();
2794           s.close();
2795
2796           out.flush();
2797           out.close();
2798
2799           out = null;
2800           buf = null;
2801           s   = null;
2802        }
2803        catch (java.io.IOException e)
2804        {
2805           System.out.println("Error:  Could not serialize acceptor vector to disk...");
2806           e.printStackTrace();
2807        }
2808
2809        trace("saveAcceptorVector()", EXIT);
2810     }
2811
2812     public void logStackedBill(int iDenomination)
2813     {
2814        trace("logStackedBill()", ENTER);
2815
2816        //           1         2
2817        // 012345678901234567890123456789
2818        // Wed Jun 21 15:52:16 EDT 2000
2819        Date now = new Date();
2820
2821
2822
2823        // Determine our entry (keyed off of month+year. e.g. 7/12/00 would be "Jul2000")
2824        String strMonthAndYear = now.toString().substring(4,7) + now.toString().substring(24,28);
2825
2826
2827
2828        boolean bLogEntryMade = false;
2829        Vector keyValueVect = null;
2830        Integer amt = null;
2831        String strVectMonthAndYear = null;
2832
2833        if (acceptorVector.size() > 0)
2834        {
2835           keyValueVect = (Vector)acceptorVector.elementAt(acceptorVector.size() - 1);
2836
2837           // See if there's already been an entry made for this month.
2838          strVectMonthAndYear = (String)keyValueVect.elementAt(0);
2839
2840           if (strVectMonthAndYear.equals(strMonthAndYear))
2841           {
2842              // Retrieve the current value (in total $ collected for that day).
2843              amt = (Integer)keyValueVect.elementAt(1);
2844              int iAmt = amt.intValue();
2845
2846              // Now, add to this value and store it back into the vector.
2847              iAmt = iAmt + iDenomination;
2848              amt = new Integer(iAmt);
2849
2850              keyValueVect.insertElementAt(amt, 1);
2851              bLogEntryMade = true;
2852           }
2853        }
2854
2855
2856        if (bLogEntryMade == false)
2857        {
2858           Vector newKeyValueVect = new Vector();
2859           amt = new Integer(iDenomination);
2860
2861           newKeyValueVect.insertElementAt(strMonthAndYear, 0);
2862           newKeyValueVect.insertElementAt(amt, 1);
2863
2864           acceptorVector.addElement(newKeyValueVect);
2865        }
2866
2867        saveAcceptorVector();
```

```
2868
2869        trace(strMonthAndYear + ": Total collected so far this month: " + amt.toString());
2870
2871        trace("logStackedBill()", EXIT);
2872     }
2873
2874     private void loadProperties()
2875     {
2876        trace("loadProperties()", ENTER);
2877
2878        // Load the properties from disk.
2879        properties = new Properties();
2880        try
2881        {
2882           File propFile = new File("MP3Jukeboxx.properties");
2883
2884           if (!propFile.exists())
2885           {
2886              logInfo("Properties file not found, using defaults...");
2887
2888              propFile = null;
2889              propFile = new File("c:/Kiosk/MP3Jukeboxx.properties");
2890           }
2891
2892           if (propFile.exists())
2893           {
2894              logInfo("Loading properties from disk...");
2895
2896              properties.load(new FileInputStream(propFile));
2897
2898              try
2899              {
2900                 // Number of outstanding credits
2901                 if (properties.getProperty("credits") != null)
2902                 {
2903                    Integer intPropCredits = new Integer(properties.getProperty("credits"));
2904
2905                    intCredits = intPropCredits;
2906                    newCredits = intCredits.intValue();
2907
2908                    // Force an update of the text field.
2909                    credits = 0;
2910
2911                    logInfo("Outstanding Credits: " + intCredits.toString());
2912                 }
2913
2914                 // Automatic play on minutes of silence.
2915                 if (properties.getProperty("RandomPlay") != null)
2916                 {
2917                    String strTemp = properties.getProperty("RandomPlay");
2918
2919                    if (strTemp.equalsIgnoreCase("false"))
2920                    {
2921                       bRandomPlay = false;
2922                    }
2923                    else
2924                    {
2925                       bRandomPlay = true;
2926                    }
2927
2928                    logInfo("Random Play=" + strTemp);
2929                 }
2930
2931                 // Interval for random play.
2932                 if (properties.getProperty("RandomPlayInterval") != null)
2933                 {
2934                    intRandomPlayInterval = new Integer(properties.getProperty("RandomPlayInterval"));
2935                    iRandomPlayInterval = intRandomPlayInterval.intValue();
2936
2937                    logInfo("Random Play Interval=" + intRandomPlayInterval.toString());
2938                 }
2939
2940                 // "Flip to Random CD" on startup.
2941                 if (properties.getProperty("FlipToRandom") != null)
2942                 {
2943                    String strTemp = properties.getProperty("FlipToRandom");
2944
2945                    if (strTemp.equalsIgnoreCase("false"))
2946                    {
2947                       bFlipToRandom = false;
2948                    }
2949                    else
2950                    {
2951                       bFlipToRandom = true;
2952                    }
2953
2954                    logInfo("FlipToRandom=" + strTemp);
2955                 }
2956
2957                 // "Show Queued" flags for songs.
```

```
2958                if (properties.getProperty("ShowQueued") != null)
2959                {
2960                   String strTemp = properties.getProperty("ShowQueued");
2961
2962                   if (strTemp.equalsIgnoreCase("false"))
2963                   {
2964                      bShowQueued = false;
2965                   }
2966                   else
2967                   {
2968                      bShowQueued = true;
2969                   }
2970
2971                   logInfo("ShowQueued=" + strTemp);
2972                }
2973
2974                // Whether or not to display a confirmation box upon song selection.
2975                if (properties.getProperty("ShowConfirmation") != null)
2976                {
2977                   String strTemp = properties.getProperty("ShowConfirmation");
2978
2979                   if (strTemp.equalsIgnoreCase("false"))
2980                   {
2981                      bShowConfirmation = false;
2982                   }
2983                   else
2984                   {
2985                      bShowConfirmation = true;
2986                   }
2987
2988                   logInfo("ShowConfirmation=" + strTemp);
2989                }
2990
2991                // Number of free songs to maintain in the queue when no paid songs are requested (to avoid
silence).
2992                if (properties.getProperty("NumberToQueue") != null)
2993                {
2994                   intNumberToQueue = new Integer(properties.getProperty("NumberToQueue"));
2995                   iNumberToQueue = intNumberToQueue.intValue();
2996
2997                   logInfo("Number of free songs to maintain in Queue=" + intNumberToQueue.toString());
2998                }
2999
3000                // Player volume (0-100)
3001                if (properties.getProperty("PlayerVolume") != null)
3002                {
3003                   intPlayerVolume = new Integer(properties.getProperty("PlayerVolume"));
3004                   iPlayerVolume = intPlayerVolume.intValue();
3005
3006                   logInfo("Player Volume=" + intPlayerVolume.toString());
3007                }
3008
3009                // Get the level 0 threshold for the list renderer.
3010                if (properties.getProperty("Level_0") != null)
3011                {
3012                   intLevel_0 = new Integer(properties.getProperty("Level_0"));
3013                   iLevel_0 = intLevel_0.intValue();
3014
3015                   logInfo("Level_0=" + intLevel_0.toString());
3016                }
3017
3018                // Get the level 1 threshold for the list renderer.
3019                if (properties.getProperty("Level_1") != null)
3020                {
3021                   intLevel_1 = new Integer(properties.getProperty("Level_1"));
3022                   iLevel_1 = intLevel_1.intValue();
3023
3024                   logInfo("Level_1=" + intLevel_1.toString());
3025                }
3026
3027                // Get the level 2 threshold for the list renderer.
3028                if (properties.getProperty("Level_2") != null)
3029                {
3030                   intLevel_2 = new Integer(properties.getProperty("Level_2"));
3031                   iLevel_2 = intLevel_2.intValue();
3032
3033                   logInfo("Level_2=" + intLevel_2.toString());
3034                }
3035
3036
3037
3038                // Get the credits per dollar value.
3039                if (properties.getProperty("CreditsPer") != null)
3040                {
3041                   intCreditsPer = new Integer(properties.getProperty("CreditsPer"));
3042                   iCreditsPer = intCreditsPer.intValue();
3043
3044                   iBonusLevel_1  = iCreditsPer * 2;   // If the user entered 2 $1.00 bills or 1 $2.00 bill
3045                   iBonusLevel_2  = iCreditsPer * 5;   // If the user entered $5.00 worth of bill(s)
3046                   iBonusLevel_3  = iCreditsPer * 10;  // If the user entered $10.00 worth of bill(s)
```

```
                iBonusLevel_4  = iCreditsPer * 20;  // If the user entered $20.00 worth of bill(s)

                logInfo("CreditsPer=" + intCreditsPer.toString());
             }

             // Get the bonus credits factor for $2.00.
             if (properties.getProperty("BonusFactor_1") != null)
             {
                intBonusFactor_1 = new Integer(properties.getProperty("BonusFactor_1"));
                iBonusFactor_1 = intBonusFactor_1.intValue();

                logInfo("BonusFactor_1(2$)=" + intBonusFactor_1.toString());
             }

             // Get the bonus credits factor for $5.00.
             if (properties.getProperty("BonusFactor_2") != null)
             {
                intBonusFactor_2 = new Integer(properties.getProperty("BonusFactor_2"));
                iBonusFactor_2 = intBonusFactor_2.intValue();

                logInfo("BonusFactor_2(5$)=" + intBonusFactor_2.toString());
             }

             // Get the bonus credits factor for $10.00.
             if (properties.getProperty("BonusFactor_3") != null)
             {
                intBonusFactor_3 = new Integer(properties.getProperty("BonusFactor_3"));
                iBonusFactor_3 = intBonusFactor_3.intValue();

                logInfo("BonusFactor_3(10$)=" + intBonusFactor_3.toString());
             }

             // Get the bonus credits factor for $20.00.
             if (properties.getProperty("BonusFactor_4") != null)
             {
                intBonusFactor_4 = new Integer(properties.getProperty("BonusFactor_4"));
                iBonusFactor_4 = intBonusFactor_4.intValue();

                logInfo("BonusFactor_4(20$)=" + intBonusFactor_4.toString());
             }

             // Get the new CD Vector Size.
             if (properties.getProperty("NewCDVectorSize") != null)
             {
                intNewCDVectorSize = new Integer(properties.getProperty("NewCDVectorSize"));
                iNewCDVectorSize = intNewCDVectorSize.intValue();

                logInfo("NewCDVectorSize=" + intNewCDVectorSize.toString());
             }

             // Get the new CD age threshold.
             if (properties.getProperty("NewCDAgeThreshold") != null)
             {
                intNewCDAgeThreshold = new Integer(properties.getProperty("NewCDAgeThreshold"));
                iNewCDAgeThreshold = intNewCDAgeThreshold.intValue();

                logInfo("NewCDAgeThreshold=" + intNewCDAgeThreshold.toString());
             }
          }
          catch(java.lang.NumberFormatException e1)
          {
             logException(e1);
             e1.printStackTrace();
          }
       }
    }
    catch (java.io.IOException e2)
    {
       logException(e2);
       e2.printStackTrace();
    }

    trace("loadProperties()", EXIT);
 }

 private void saveProperties()
 {
    trace("saveProperties()", ENTER);

    try
    {
       properties.put("credits", intCredits.toString());


       if (bRandomPlay == true)
          properties.put("RandomPlay", "true");
       else
          properties.put("RandomPlay", "false");

       properties.put("RandomPlayInterval", intRandomPlayInterval.toString());
```

```
3138
3139            if (bFlipToRandom == true)
3140               properties.put("FlipToRandom", "true");
3141            else
3142               properties.put("FlipToRandom", "false");
3143
3144
3145            if (bShowQueued == true)
3146               properties.put("ShowQueued", "true");
3147            else
3148               properties.put("ShowQueued", "false");
3149
3150
3151            if (bShowConfirmation == true)
3152               properties.put("ShowConfirmation", "true");
3153            else
3154               properties.put("ShowConfirmation", "false");
3155
3156
3157            properties.put("NumberToQueue", intNumberToQueue.toString());
3158            properties.put("PlayerVolume", intPlayerVolume.toString());
3159
3160
3161            properties.put("Level_0", intLevel_0.toString());
3162            properties.put("Level_1", intLevel_1.toString());
3163            properties.put("Level_2", intLevel_2.toString());
3164
3165
3166            properties.put("CreditsPer", intCreditsPer.toString());
3167            properties.put("BonusFactor_1", intBonusFactor_1.toString());
3168            properties.put("BonusFactor_2", intBonusFactor_2.toString());
3169            properties.put("BonusFactor_3", intBonusFactor_3.toString());
3170            properties.put("BonusFactor_4", intBonusFactor_4.toString());
3171
3172
3173            properties.put("NewCDVectorSize", intNewCDVectorSize.toString());
3174            properties.put("NewCDAgeThreshold", intNewCDAgeThreshold.toString());
3175
3176            // Save the properties to disk.
3177            properties.store(new FileOutputStream("MP3Jukeboxx.properties"), "description");
3178         }
3179         catch (java.io.IOException e)
3180         {
3181            try
3182            {
3183               properties.store(new FileOutputStream("c:/Kiosk/MP3Jukeboxx.properties"), "description");
3184            }
3185            catch(java.io.IOException e2)
3186            {
3187               System.out.println("ERROR: Couldn't save properties file!");
3188            }
3189         }
3190
3191         trace("saveProperties()", EXIT);
3192      }
3193
3194      private void initTree()
3195      {
3196         trace("initTree()", ENTER);
3197
3198         treeModel = treeMgr.createTree();
3199         tree.setModel(treeModel);
3200
3201         // Expand all the nodes in the tree before displaying it.
3202         int row = 0;
3203         while (row <= tree.getRowCount())
3204         {
3205            tree.expandRow(row);
3206            row++;
3207         }
3208
3209         // Set the RandomMode
3210         treeMgr.setRandomMode(true);
3211
3212         // Set the total number of rows in the tree (used for generating a random element)
3213         treeMgr.setRowCount(tree.getRowCount());
3214
3215
3216         if ((tree.getRowCount() > 1))
3217         {
3218            tree.setVisible(true);
3219
3220            // Reset the queued flags as the counts are regenerated as the songs are re-added to the song queue.
3221            treeMgr.resetAllQueuedCnts();
3222
3223            // Load the playlist via what was serialized at last exit.
3224            File file = new File("MP3Jukeboxx.PL");
3225            if (!file.exists())
3226            {
3227               file = null;
```

```
3228             file = new File("c:/kiosk/MP3Jukeboxx.PL");
3229          }
3230
3231          if (file.exists())
3232          {
3233             logInfo("Loading outstanding playlist from disk...");
3234
3235             playerMgr.flushPlayList();
3236
3237             Vector vector = playerMgr.loadPlayList(file);
3238             String mp3 = null;
3239             PlayListEntry tmpObject;
3240
3241             for (Enumeration e = vector.elements(); e.hasMoreElements(); )
3242             {
3243                mp3 = e.nextElement().toString();
3244
3245                File plfile = new File(mp3);
3246                if (plfile.exists())
3247                {
3248                   tmpObject = treeMgr.getPlayListObjForSong(mp3);
3249                   if (tmpObject instanceof PlayListEntry)
3250                   {
3251                      playerMgr.addToPlayList(tmpObject);
3252
3253                      logInfo("Adding: " + tmpObject.toString() + " to the playlist");
3254                   }
3255                }
3256                else
3257                {
3258                   logInfo("Playlist entry not found: " + mp3);
3259                }
3260             }
3261          }
3262
3263          // Need to have a song selected to have the buttons enabled.
3264          treeMgr.selectInitialSong(tree);
3265
3266          bIsAppFunctional = true;
3267       }
3268       else
3269       {
3270          logInfo("No songs found in the tree, displaying Add Path Dialog");
3271
3272          // Disable everything until MP3s are added to the jukebox tree.
3273          bIsAppFunctional = false;
3274          //disableFunctionality();
3275
3276          this.setEnabled(true);
3277          // Bring up the Add Drive/Dir window.
3278          menuAddDir_doWork();
3279       }
3280       this.setEnabled(true);
3281
3282       trace("initTree()", EXIT);
3283    }
3284
3285    private void initGui()
3286    {
3287       trace("initGui()", ENTER);
3288
3289       cardPanel = new JPanel();
3290       cardPanel.setBounds(0,3,1024,600);
3291       cardPanel.setForeground(Color.white);
3292       cardPanel.setBackground(Color.black);
3293       cardMgr = new CardLayout();
3294       cardPanel.setLayout(cardMgr);
3295
3296
3297       userPanel = new JPanel();
3298       userPanel.setBounds(0,3,1024,600);
3299       userPanel.setForeground(Color.white);
3300       userPanel.setBackground(Color.black);
3301       userCardMgr = new CardLayout();
3302       userPanel.setLayout(userCardMgr);
3303
3304
3305       bottomPanel = new JPanel();
3306       bottomPanel.setBorder(new LineBorder(Color.white, 1));
3307       bottomPanel.setBounds(1,603,1022,167);
3308       bottomPanel.setForeground(Color.white);
3309       bottomPanel.setBackground(Color.black);
3310       bottomPanel.setLayout(null);
3311
3312       treeViewBtn = new JButton("");
3313       treeViewBtn.setBounds(1,1,40,40);
3314       treeViewBtn.addActionListener(this);
3315       treeViewBtn.setForeground(Color.white);
3316       treeViewBtn.setBackground(Color.black);
3317       treeViewBtn.setBorderPainted(false);
```

```
3318        treeViewBtn.setFocusPainted(false);
3319        bottomPanel.add(treeViewBtn);
3320
3321        srchBtn = new JButton();
3322        srchBtn.setBounds(73,11,125,50);
3323        srchBtn.setBorderPainted(false);
3324        srchBtn.setFocusPainted(false);
3325        srchBtn.setIcon(loadIcon("images/search.gif"));
3326        srchBtn.setDisabledIcon(loadIcon("images/searchdisabled.gif"));
3327        srchBtn.setPressedIcon(loadIcon("images/searchpressed.gif"));
3328        srchBtn.addActionListener(this);
3329        bottomPanel.add(srchBtn);
3330
3331        tableBtn = new JButton();
3332        tableBtn.setBounds(199,11,125,50);
3333        tableBtn.setBorderPainted(false);
3334        tableBtn.setFocusPainted(false);
3335        tableBtn.setIcon(loadIcon("images/popular.gif"));
3336        tableBtn.setDisabledIcon(loadIcon("images/populardisabled.gif"));
3337        tableBtn.setPressedIcon(loadIcon("images/popularpressed.gif"));
3338        tableBtn.addActionListener(this);
3339        bottomPanel.add(tableBtn);
3340
3341        genreBtn = new JButton();
3342        genreBtn.setBounds(325,11,125,50);
3343        genreBtn.setBorderPainted(false);
3344        genreBtn.setFocusPainted(false);
3345        genreBtn.setIcon(loadIcon("images/genre.gif"));
3346        genreBtn.setDisabledIcon(loadIcon("images/genredisabled.gif"));
3347        genreBtn.setPressedIcon(loadIcon("images/genrepressed.gif"));
3348        genreBtn.addActionListener(this);
3349        bottomPanel.add(genreBtn);
3350
3351        showCurrentBtn = new JButton();
3352        showCurrentBtn.setBounds(467,11,50,50);
3353        showCurrentBtn.setBorderPainted(false);
3354        showCurrentBtn.setFocusPainted(false);
3355        showCurrentBtn.setIcon(loadIcon("images/showcurrent.gif"));
3356        showCurrentBtn.setDisabledIcon(loadIcon("images/showcurrentdisabled.gif"));
3357        showCurrentBtn.setPressedIcon(loadIcon("images/showcurrentpressed.gif"));
3358        showCurrentBtn.addActionListener(this);
3359        showCurrentBtn.setEnabled(false);
3360        bottomPanel.add(showCurrentBtn);
3361
3362
3363        topBtn = new JButton();
3364        topBtn.setBounds(30,64,50,50);
3365        topBtn.setEnabled(false);
3366        topBtn.addActionListener(this);
3367        topBtn.setBorderPainted(false);
3368        topBtn.setFocusPainted(false);
3369        topBtn.setIcon(loadIcon("images/top.gif"));
3370        topBtn.setDisabledIcon(loadIcon("images/topdisabled.gif"));
3371        topBtn.setPressedIcon(loadIcon("images/toppressed.gif"));
3372        bottomPanel.add(topBtn);
3373
3374        prevGenreBtn = new JButton();
3375        prevGenreBtn.setBounds(30,115,50,50);
3376        prevGenreBtn.setEnabled(false);
3377        prevGenreBtn.addActionListener(this);
3378        prevGenreBtn.setBorderPainted(false);
3379        prevGenreBtn.setFocusPainted(false);
3380        prevGenreBtn.setIcon(loadIcon("images/prevgenre.gif"));
3381        prevGenreBtn.setDisabledIcon(loadIcon("images/prevgenredisabled.gif"));
3382        prevGenreBtn.setPressedIcon(loadIcon("images/prevgenrepressed.gif"));
3383        prevGenreBtn.setEnabled(false);
3384        bottomPanel.add(prevGenreBtn);
3385
3386        prevPageBtn = new JButton();
3387        prevPageBtn.setBounds(103,77,150,75);
3388        prevPageBtn.setEnabled(false);
3389        prevPageBtn.addActionListener(this);
3390        prevPageBtn.setBorderPainted(false);
3391        prevPageBtn.setFocusPainted(false);
3392        prevPageBtn.setIcon(loadIcon("images/prevpage.gif"));
3393        prevPageBtn.setDisabledIcon(loadIcon("images/prevpagedisabled.gif"));
3394        prevPageBtn.setPressedIcon(loadIcon("images/prevpagepressed.gif"));
3395        bottomPanel.add(prevPageBtn);
3396
3397        nextPageBtn = new JButton();
3398        nextPageBtn.setBounds(267,77,150,75);
3399        nextPageBtn.addActionListener(this);
3400        nextPageBtn.setBorderPainted(false);
3401        nextPageBtn.setFocusPainted(false);
3402        nextPageBtn.setIcon(loadIcon("images/nextpage.gif"));
3403        nextPageBtn.setDisabledIcon(loadIcon("images/nextpagedisabled.gif"));
3404        nextPageBtn.setPressedIcon(loadIcon("images/nextpagepressed.gif"));
3405        bottomPanel.add(nextPageBtn);
3406
3407        nextGenreBtn = new JButton();
```

```
3408        nextGenreBtn.setBounds(440,64,50,50);
3409        nextGenreBtn.addActionListener(this);
3410        nextGenreBtn.setBorderPainted(false);
3411        nextGenreBtn.setFocusPainted(false);
3412        nextGenreBtn.setIcon(loadIcon("images/nextgenre.gif"));
3413        nextGenreBtn.setDisabledIcon(loadIcon("images/nextgenredisabled.gif"));
3414        nextGenreBtn.setPressedIcon(loadIcon("images/nextgenrepressed.gif"));
3415        bottomPanel.add(nextGenreBtn);
3416
3417        btmBtn = new JButton();
3418        btmBtn.setBounds(440,115,50,50);
3419        btmBtn.addActionListener(this);
3420        btmBtn.setBorderPainted(false);
3421        btmBtn.setFocusPainted(false);
3422        btmBtn.setIcon(loadIcon("images/btm.gif"));
3423        btmBtn.setDisabledIcon(loadIcon("images/btmdisabled.gif"));
3424        btmBtn.setPressedIcon(loadIcon("images/btmpressed.gif"));
3425        bottomPanel.add(btmBtn);
3426
3427
3428
3429        nowPlayingLabel = new JLabel("Selection Currently Playing:");
3430        nowPlayingLabel.setBounds(525,10,280,15);
3431        nowPlayingLabel.setForeground(Color.yellow);
3432        nowPlayingLabel.setBackground(Color.black);
3433        nowPlayingLabel.setFont(new Font("SansSerif", Font.BOLD, 12));
3434        bottomPanel.add(nowPlayingLabel);
3435
3436        nowPlayingTxtField = new JTextField("");
3437        nowPlayingTxtField.setBounds(525,27,410,15);
3438        nowPlayingTxtField.setHorizontalAlignment(JTextField.LEFT);
3439        nowPlayingTxtField.setForeground(Color.white);
3440        nowPlayingTxtField.setBackground(Color.black);
3441        nowPlayingTxtField.setFont(new Font("SansSerif", Font.BOLD, 12));
3442        nowPlayingTxtField.setEditable(false);
3443        nowPlayingTxtField.setAutoscrolls(false);
3444        bottomPanel.add(nowPlayingTxtField);
3445
3446
3447
3448        totalCDsLabel = new JLabel("Total CDs:");
3449        totalCDsLabel.setBounds(950,10,60,15);
3450        totalCDsLabel.setForeground(Color.yellow);
3451        totalCDsLabel.setBackground(Color.black);
3452        totalCDsLabel.setFont(new Font("SansSerif", Font.BOLD, 12));
3453        bottomPanel.add(totalCDsLabel);
3454
3455        totalCDsTxtField = new JTextField("");
3456        totalCDsTxtField.setBounds(950,27,55,15);
3457        totalCDsTxtField.setHorizontalAlignment(JTextField.RIGHT);
3458        totalCDsTxtField.setForeground(Color.white);
3459        totalCDsTxtField.setBackground(Color.black);
3460        totalCDsTxtField.setFont(new Font("SansSerif", Font.BOLD, 12));
3461        totalCDsTxtField.addMouseListener(this);
3462        bottomPanel.add(totalCDsTxtField);
3463
3464
3465
3466        creditsLabel = new JLabel("Selections Remaining:");
3467        creditsLabel.setBounds(890,50,150,15);
3468        creditsLabel.setForeground(Color.yellow);
3469        creditsLabel.setBackground(Color.black);
3470        creditsLabel.setFont(new Font("SansSerif", Font.BOLD, 12));
3471        bottomPanel.add(creditsLabel);
3472
3473        creditsTxtField = new JTextField(intCredits.toString());
3474        creditsTxtField.setBounds(925,67,51,15);
3475        creditsTxtField.setHorizontalAlignment(JTextField.RIGHT);
3476        creditsTxtField.setForeground(Color.white);
3477        creditsTxtField.setBackground(Color.black);
3478        creditsTxtField.setFont(new Font("SansSerif", Font.BOLD, 12));
3479        bottomPanel.add(creditsTxtField);
3480
3481
3482
3483        selectionLabel = new JLabel("Selection Being Made:");
3484        selectionLabel.setBounds(890,90,150,15);
3485        selectionLabel.setForeground(Color.yellow);
3486        selectionLabel.setBackground(Color.black);
3487        selectionLabel.setFont(new Font("SansSerif", Font.BOLD, 12));
3488        bottomPanel.add(selectionLabel);
3489
3490        selectionTxtField = new JTextField("");
3491        selectionTxtField.setBounds(925,107,51,15);
3492        selectionTxtField.setHorizontalAlignment(JTextField.RIGHT);
3493        selectionTxtField.setForeground(Color.white);
3494        selectionTxtField.setBackground(Color.black);
3495        selectionTxtField.setFont(new Font("SansSerif", Font.BOLD, 12));
3496        bottomPanel.add(selectionTxtField);
3497
```

```
3499
3500        visibleCDsLabel = new JLabel("Visible CDs:");
3501        visibleCDsLabel.setBounds(935,130,80,15);
3502        visibleCDsLabel.setForeground(Color.yellow);
3503        visibleCDsLabel.setBackground(Color.black);
3504        visibleCDsLabel.setFont(new Font("SansSerif", Font.BOLD, 12));
3505        bottomPanel.add(visibleCDsLabel);
3506
3507        visibleCDsTxtField = new JTextField("");
3508        visibleCDsTxtField.setBounds(935,147,70,15);
3509        visibleCDsTxtField.setHorizontalAlignment(JTextField.CENTER);
3510        visibleCDsTxtField.setForeground(Color.white);
3511        visibleCDsTxtField.setBackground(Color.black);
3512        visibleCDsTxtField.setFont(new Font("SansSerif", Font.BOLD, 12));
3513        bottomPanel.add(visibleCDsTxtField);
3514
3515
3516
3517        strText = null;
3518        strText = selectionTxtField.getText();
3519
3520        btn_1 = new JButton();
3521        btn_1.setBounds(515,65,50,50);
3522        btn_1.addActionListener(this);
3523        btn_1.setBorderPainted(false);
3524        btn_1.setFocusPainted(false);
3525        btn_1.setIcon(loadIcon("images/one.gif"));
3526        btn_1.setPressedIcon(loadIcon("images/onepressed.gif"));
3527        btn_1.setDisabledIcon(loadIcon("images/onedisabled.gif"));
3528        bottomPanel.add(btn_1);
3529
3530        btn_2 = new JButton();
3531        btn_2.setBounds(565,65,50,50);
3532        btn_2.addActionListener(this);
3533        btn_2.setBorderPainted(false);
3534        btn_2.setFocusPainted(false);
3535        btn_2.setIcon(loadIcon("images/two.gif"));
3536        btn_2.setPressedIcon(loadIcon("images/twopressed.gif"));
3537        btn_2.setDisabledIcon(loadIcon("images/twodisabled.gif"));
3538        bottomPanel.add(btn_2);
3539
3540        btn_3 = new JButton();
3541        btn_3.setBounds(615,65,50,50);
3542        btn_3.addActionListener(this);
3543        btn_3.setBorderPainted(false);
3544        btn_3.setFocusPainted(false);
3545        btn_3.setIcon(loadIcon("images/three.gif"));
3546        btn_3.setPressedIcon(loadIcon("images/threepressed.gif"));
3547        btn_3.setDisabledIcon(loadIcon("images/threedisabled.gif"));
3548        bottomPanel.add(btn_3);
3549
3550        btn_4 = new JButton();
3551        btn_4.setBounds(665,65,50,50);
3552        btn_4.addActionListener(this);
3553        btn_4.setBorderPainted(false);
3554        btn_4.setFocusPainted(false);
3555        btn_4.setIcon(loadIcon("images/four.gif"));
3556        btn_4.setPressedIcon(loadIcon("images/fourpressed.gif"));
3557        btn_4.setDisabledIcon(loadIcon("images/fourdisabled.gif"));
3558        bottomPanel.add(btn_4);
3559
3560        btn_5 = new JButton();
3561        btn_5.setBounds(715,65,50,50);
3562        btn_5.addActionListener(this);
3563        btn_5.setBorderPainted(false);
3564        btn_5.setFocusPainted(false);
3565        btn_5.setIcon(loadIcon("images/five.gif"));
3566        btn_5.setPressedIcon(loadIcon("images/fivepressed.gif"));
3567        btn_5.setDisabledIcon(loadIcon("images/fivedisabled.gif"));
3568        bottomPanel.add(btn_5);
3569
3570        cancelBtn = new JButton();
3571        cancelBtn.setBounds(765,65,110,50);
3572        cancelBtn.addActionListener(this);
3573        cancelBtn.setBorderPainted(false);
3574        cancelBtn.setFocusPainted(false);
3575        cancelBtn.setIcon(loadIcon("images/cancel.gif"));
3576        cancelBtn.setPressedIcon(loadIcon("images/cancelpressed.gif"));
3577        cancelBtn.setDisabledIcon(loadIcon("images/canceldisabled.gif"));
3578        bottomPanel.add(cancelBtn);
3579
3580        btn_6 = new JButton();
3581        btn_6.setBounds(515,115,50,50);
3582        btn_6.addActionListener(this);
3583        btn_6.setBorderPainted(false);
3584        btn_6.setFocusPainted(false);
3585        btn_6.setIcon(loadIcon("images/six.gif"));
3586        btn_6.setPressedIcon(loadIcon("images/sixpressed.gif"));
3587        btn_6.setDisabledIcon(loadIcon("images/sixdisabled.gif"));
3588        bottomPanel.add(btn_6);
```

```
3590        btn_7 = new JButton();
3591        btn_7.setBounds(565,115,50,50);
3592        btn_7.addActionListener(this);
3593        btn_7.setBorderPainted(false);
3594        btn_7.setFocusPainted(false);
3595        btn_7.setIcon(loadIcon("images/seven.gif"));
3596        btn_7.setPressedIcon(loadIcon("images/sevenpressed.gif"));
3597        btn_7.setDisabledIcon(loadIcon("images/sevendisabled.gif"));
3598        bottomPanel.add(btn_7);
3599
3600        btn_8 = new JButton();
3601        btn_8.setBounds(615,115,50,50);
3602        btn_8.addActionListener(this);
3603        btn_8.setBorderPainted(false);
3604        btn_8.setFocusPainted(false);
3605        btn_8.setIcon(loadIcon("images/eight.gif"));
3606        btn_8.setPressedIcon(loadIcon("images/eightpressed.gif"));
3607        btn_8.setDisabledIcon(loadIcon("images/eightdisabled.gif"));
3608        bottomPanel.add(btn_8);
3609
3610        btn_9 = new JButton();
3611        btn_9.setBounds(665,115,50,50);
3612        btn_9.addActionListener(this);
3613        btn_9.setBorderPainted(false);
3614        btn_9.setFocusPainted(false);
3615        btn_9.setIcon(loadIcon("images/nine.gif"));
3616        btn_9.setPressedIcon(loadIcon("images/ninepressed.gif"));
3617        btn_9.setDisabledIcon(loadIcon("images/ninedisabled.gif"));
3618        bottomPanel.add(btn_9);
3619
3620        btn_0 = new JButton();
3621        btn_0.setBounds(715,115,50,50);
3622        btn_0.addActionListener(this);
3623        btn_0.setBorderPainted(false);
3624        btn_0.setFocusPainted(false);
3625        btn_0.setIcon(loadIcon("images/zero.gif"));
3626        btn_0.setPressedIcon(loadIcon("images/zeropressed.gif"));
3627        btn_0.setDisabledIcon(loadIcon("images/zerodisabled.gif"));
3628        bottomPanel.add(btn_0);
3629
3630        enterBtn = new JButton();
3631        enterBtn.setBounds(765,115,110,50);
3632        enterBtn.addActionListener(this);
3633        enterBtn.setBorderPainted(false);
3634        enterBtn.setFocusPainted(false);
3635        enterBtn.setIcon(loadIcon("images/enter.gif"));
3636        enterBtn.setPressedIcon(loadIcon("images/enterpressed.gif"));
3637        enterBtn.setDisabledIcon(loadIcon("images/enterdisabled.gif"));
3638        bottomPanel.add(enterBtn);
3639
3640
3641        // Make the following flag true to cause the action.
3642        bButtonsEnabled = true;
3643        toggleButtons(false);
3644
3645
3646        // Tree Panel.
3647        adminPanel = new JPanel();
3648        adminPanel.setForeground(Color.white);
3649        adminPanel.setBackground(Color.black);
3650        adminPanel.setBounds(0,0,1024,600);
3651        adminPanel.setLayout(null);
3652
3653        treeLabel = new JLabel("Jukebox Tree:");
3654        treeLabel.setBounds(70,2,100,17);
3655        treeLabel.setFont(new Font("SansSerif", Font.BOLD, 12));
3656        treeLabel.setForeground(Color.yellow);
3657        adminPanel.add(treeLabel);
3658
3659        playlistLabel = new JLabel("Song Queue:");
3660        playlistLabel.setBounds(523,2,100,17);
3661        playlistLabel.setFont(new Font("SansSerif", Font.BOLD, 12));
3662        playlistLabel.setForeground(Color.yellow);
3663        adminPanel.add(playlistLabel);
3664
3665        billStatsLabel = new JLabel("Acceptor Statistics:");
3666        billStatsLabel.setBounds(300,454,150,17);
3667        billStatsLabel.setFont(new Font("SansSerif", Font.BOLD, 12));
3668        billStatsLabel.setForeground(Color.yellow);
3669        adminPanel.add(billStatsLabel);
3670
3671        adminLogLabel = new JLabel("Log File History:");
3672        adminLogLabel.setBounds(523,454,100,17);
3673        adminLogLabel.setFont(new Font("SansSerif", Font.BOLD, 12));
3674        adminLogLabel.setForeground(Color.yellow);
3675        adminPanel.add(adminLogLabel);
3676
3677
3678        treeModel = new DefaultTreeModel(new DefaultMutableTreeNode("Root"));
3679        tree = new JTree(treeModel);
```

```
3681        treeScrollPane = new JScrollPane();
3682        treeScrollPane.setBounds(63,20,448,432);
3683        treeScrollPane.getViewport().add(tree);
3684
3685        JScrollBar horizontal = treeScrollPane.getHorizontalScrollBar();
3686        horizontal.setPreferredSize(new Dimension(horizontal.getWidth(),25));
3687
3688        JScrollBar vertical = treeScrollPane.getVerticalScrollBar();
3689        vertical.setPreferredSize(new Dimension(25,vertical.getHeight()));
3690
3691        tree.setVisible(false);
3692        tree.setLargeModel(true);
3693        tree.setBounds(0,0,510,594);
3694        tree.setForeground(Color.white);
3695        tree.setBackground(Color.black);
3696        tree.setBorder(new LineBorder(Color.white, 1));
3697        DefaultTreeSelectionModel selectionModel = new DefaultTreeSelectionModel();
3698        selectionModel.setSelectionMode(javax.swing.tree.DefaultTreeSelectionModel.SINGLE_TREE_SELECTION);
3699        tree.setSelectionModel(selectionModel);
3700        tree.setCellRenderer(new MyRenderer(iLevel_0, iLevel_1, iLevel_2));
3701        adminPanel.add(treeScrollPane);
3702
3703
3704
3705        billStats = createBillStatsTable();
3706        billStatsScrollPane = new JScrollPane();
3707        billStatsScrollPane.setForeground(Color.white);
3708        billStatsScrollPane.setBackground(Color.black);
3709        billStatsScrollPane.setBounds(290,470,200,105);
3710                  billStatsScrollPane.getViewport().add(billStats);
3711        adminPanel.add(billStatsScrollPane);
3712
3713
3714
3715        playlistVector = playerMgr.getPlayListVector();
3716        playlistList = new JList(playlistVector);
3717        playlistList.setForeground(Color.white);
3718        playlistList.setBackground(Color.black);
3719        playlistList.setBounds(0,0,450,589);
3720        playlistList.addListSelectionListener(
3721           new ListSelectionListener()
3722           {
3723              public void valueChanged(javax.swing.event.ListSelectionEvent event)
3724              {
3725                 int i = playlistList.getMaxSelectionIndex();
3726
3727                 if (i >= 0)
3728                 {
3729
3730                 }
3731              }
3732           });
3733
3734
3735        playlistScrollPane = new JScrollPane();
3736        playlistScrollPane.setBounds(513,20,451,432);
3737        playlistScrollPane.getViewport().add(playlistList);
3738
3739        JScrollBar plHorizontal = playlistScrollPane.getHorizontalScrollBar();
3740        plHorizontal.setPreferredSize(new Dimension(plHorizontal.getWidth(),25));
3741
3742        JScrollBar plVertical = playlistScrollPane.getVerticalScrollBar();
3743        plVertical.setPreferredSize(new Dimension(25,plVertical.getHeight()));
3744
3745        adminPanel.add(playlistScrollPane);
3746
3747
3748        adminNextBtn = new JButton(loadIcon("images/playnext.gif"));
3749        adminNextBtn.setBounds(970,5,48,48);
3750        adminNextBtn.setEnabled(true);
3751        adminNextBtn.addActionListener(this);
3752        adminPanel.add(adminNextBtn);
3753
3754
3755        adminPauseBtn = new JButton(loadIcon("images/pause.gif"));
3756        adminPauseBtn.setBounds(970,55,48,48);
3757        adminPauseBtn.setEnabled(true);
3758        adminPauseBtn.addActionListener(this);
3759        adminPanel.add(adminPauseBtn);
3760
3761
3762        adminPlayBtn = new JButton(loadIcon("images/playnow.gif"));
3763        adminPlayBtn.setBounds(970,105,48,48);
3764        adminPlayBtn.setEnabled(true);
3765        adminPlayBtn.addActionListener(this);
3766        adminPanel.add(adminPlayBtn);
3767
3768
3769        adminMoveUpBtn = new JButton(loadIcon("images/moveup.gif"));
3770        adminMoveUpBtn.setBounds(970,155,48,48);
```

```
3771        adminMoveUpBtn.setEnabled(true);
3772        adminMoveUpBtn.addActionListener(this);
3773        adminPanel.add(adminMoveUpBtn);
3774
3775
3776        adminMoveDnBtn = new JButton(loadIcon("images/movedown.gif"));
3777        adminMoveDnBtn.setBounds(970,205,48,48);
3778        adminMoveDnBtn.setEnabled(true);
3779        adminMoveDnBtn.addActionListener(this);
3780        adminPanel.add(adminMoveDnBtn);
3781
3782
3783        adminRemoveBtn = new JButton(loadIcon("images/remove.gif"));
3784        adminRemoveBtn.setBounds(970,255,48,48);
3785        adminRemoveBtn.setEnabled(true);
3786        adminRemoveBtn.addActionListener(this);
3787        adminPanel.add(adminRemoveBtn);
3788
3789
3790        ownerIncrementBtn = new JButton(loadIcon("images/increment.gif"));
3791        ownerIncrementBtn.setBounds(970,305,48,48);
3792        ownerIncrementBtn.setEnabled(true);
3793        ownerIncrementBtn.setVisible(false);
3794        ownerIncrementBtn.addActionListener(this);
3795        adminPanel.add(ownerIncrementBtn);
3796
3797
3798        ownerDecrementBtn = new JButton(loadIcon("images/decrement.gif"));
3799        ownerDecrementBtn.setBounds(970,355,48,48);
3800        ownerDecrementBtn.setEnabled(true);
3801        ownerDecrementBtn.setVisible(false);
3802        ownerDecrementBtn.addActionListener(this);
3803        adminPanel.add(ownerDecrementBtn);
3804
3805
3806        ownerAddPathBtn = new JButton(loadIcon("images/addto.gif"));
3807        ownerAddPathBtn.setBounds(970,405,48,48);
3808        ownerAddPathBtn.setEnabled(true);
3809        ownerAddPathBtn.setVisible(false);
3810        ownerAddPathBtn.addActionListener(this);
3811        adminPanel.add(ownerAddPathBtn);
3812
3813
3814        ownerDeleteFromDiskBtn = new JButton(loadIcon("images/deleteall.gif"));
3815        ownerDeleteFromDiskBtn.setBounds(10,5,48,48);
3816        ownerDeleteFromDiskBtn.setEnabled(true);
3817        ownerDeleteFromDiskBtn.setVisible(false);
3818        ownerDeleteFromDiskBtn.addActionListener(this);
3819        adminPanel.add(ownerDeleteFromDiskBtn);
3820
3821        ownerAddNodeToQBtn = new JButton(loadIcon("images/addall.gif"));
3822        ownerAddNodeToQBtn.setBounds(10,55,48,48);
3823        ownerAddNodeToQBtn.setEnabled(true);
3824        ownerAddNodeToQBtn.setVisible(false);
3825        ownerAddNodeToQBtn.addActionListener(this);
3826        adminPanel.add(ownerAddNodeToQBtn);
3827
3828        ownerRemNodeFromQBtn = new JButton(loadIcon("images/removeall.gif"));
3829        ownerRemNodeFromQBtn.setBounds(10,105,48,48);
3830        ownerRemNodeFromQBtn.setEnabled(true);
3831        ownerRemNodeFromQBtn.setVisible(false);
3832        ownerRemNodeFromQBtn.addActionListener(this);
3833        adminPanel.add(ownerRemNodeFromQBtn);
3834
3835        ownerResetTreeBtn = new JButton(loadIcon("images/resetall.gif"));
3836        ownerResetTreeBtn.setBounds(10,155,48,48);
3837        ownerResetTreeBtn.setEnabled(true);
3838        ownerResetTreeBtn.setVisible(false);
3839        ownerResetTreeBtn.addActionListener(this);
3840        adminPanel.add(ownerResetTreeBtn);
3841
3842
3843        adminPlayerVolumeLabel = new JLabel("Volume:");
3844        adminPlayerVolumeLabel.setBounds(7,377,53,15);
3845        adminPlayerVolumeLabel.setFont(new Font("SansSerif", Font.BOLD, 12));
3846        adminPlayerVolumeLabel.setForeground(Color.yellow);
3847        adminPanel.add(adminPlayerVolumeLabel);
3848        adminPlayerVolumeSB = new SpinButton(100,0,iPlayerVolume);
3849        adminPlayerVolumeSB.setLocation(new Point(1,389));
3850        adminPanel.add(adminPlayerVolumeSB);
3851
3852
3853        ownerNumToQueueLabel = new JLabel("Num. Free Songs to Keep Queued:");
3854        ownerNumToQueueLabel.setBounds(5,455,200,15);
3855        ownerNumToQueueLabel.setFont(new Font("SansSerif", Font.BOLD, 12));
3856        ownerNumToQueueLabel.setForeground(Color.yellow);
3857        adminPanel.add(ownerNumToQueueLabel);
3858        ownerNumToQueueSB = new SpinButton(25,0,iNumberToQueue);
3859        ownerNumToQueueSB.setLocation(new Point(1,467));
3860        adminPanel.add(ownerNumToQueueSB);
```

```
3861
3862
3863        adminRandomIntervalLabel = new JLabel("Random Play Time Interval:");
3864        adminRandomIntervalLabel.setBounds(5,533,200,15);
3865        adminRandomIntervalLabel.setFont(new Font("SansSerif", Font.BOLD, 12));
3866        adminRandomIntervalLabel.setForeground(Color.yellow);
3867        adminPanel.add(adminRandomIntervalLabel);
3868        adminRandomIntervalSB = new SpinButton(120,20,iRandomPlayInterval);
3869        adminRandomIntervalSB.setLocation(new Point(1,545));
3870        adminPanel.add(adminRandomIntervalSB);
3871        if (bRandomPlay == false)
3872        {
3873           adminRandomIntervalLabel.setVisible(false);
3874           adminRandomIntervalSB.setVisible(false);
3875        }
3876
3877
3878        adminShowQueuedCB = new JCheckBox("Show Queued Songs To User", bShowQueued);
3879        adminShowQueuedCB.setBounds(65,482,200,25);
3880        adminShowQueuedCB.addActionListener(this);
3881        adminShowQueuedCB.setFont(new Font("SansSerif", Font.BOLD, 12));
3882        adminShowQueuedCB.setForeground(Color.yellow);
3883        adminShowQueuedCB.setBackground(Color.black);
3884        adminPanel.add(adminShowQueuedCB);
3885
3886        adminRandomPlayCB = new JCheckBox("Play Random Free Songs", bRandomPlay);
3887        adminRandomPlayCB.setBounds(65,560,200,25);
3888        adminRandomPlayCB.addActionListener(this);
3889        adminRandomPlayCB.setFont(new Font("SansSerif", Font.BOLD, 12));
3890        adminRandomPlayCB.setForeground(Color.yellow);
3891        adminRandomPlayCB.setBackground(Color.black);
3892        adminPanel.add(adminRandomPlayCB);
3893
3894        adminShowConfirmationCB = new JCheckBox("Show Selection Confirmations To User", bShowConfirmation);
3895        adminShowConfirmationCB.setBounds(275,573,275,25);
3896        adminShowConfirmationCB.addActionListener(this);
3897        adminShowConfirmationCB.setFont(new Font("SansSerif", Font.BOLD, 12));
3898        adminShowConfirmationCB.setForeground(Color.yellow);
3899        adminShowConfirmationCB.setBackground(Color.black);
3900        adminPanel.add(adminShowConfirmationCB);
3901
3902
3903        adminLogTextArea = new JTextArea();
3904        adminLogScrollPane = new JScrollPane();
3905        adminLogScrollPane.setBounds(513,470,451,105);
3906        adminLogScrollPane.getViewport().add(adminLogTextArea);
3907        adminPanel.add(adminLogScrollPane);
3908
3909        trace("initGui()", EXIT);
3910     }
3911
3912     void menuAddDir_doWork()
3913     {
3914        trace("menuAddDir_doWork()", ENTER);
3915
3916        Rectangle rect = this.getBounds();
3917        int x = ((rect.width / 2) - 160);
3918        int y = ((rect.height / 2) - 120);
3919
3920        boolean bState = true;
3921        while (bState)
3922        {
3923           JFrame addPathFrame = new JFrame();
3924           AddPathDialog addPathDialog = new AddPathDialog(addPathFrame, "Scan Drive(s) for Songs", true, x, y);
3925
3926           // While the user is still adding drives/directories, display the dialog.
3927           bState = addPathDialog.getState();
3928           if (bState)
3929           {
3930              int iBeforeCnt = treeMgr.getIntSongCount();
3931              File file;
3932
3933
3934              for (Enumeration e = addPathDialog.getVector().elements(); e.hasMoreElements(); )
3935              {
3936                 file = (File)e.nextElement();
3937
3938                 logInfo("Scanning: " + file.toString());
3939                 treeMgr.addPathToTree(tree, file);
3940                 logInfo("Done scanning: " + file.toString());
3941              }
3942              this.setEnabled(true);
3943
3944
3945              // Set the total number of rows in the tree.
3946              treeMgr.expandTree(tree);
3947              treeMgr.setRowCount(tree.getRowCount());
3948
3949
3950              // Make the tree visible if there's at least one node in it.
```

```
3951                if (tree.getRowCount() > 1)
3952                   tree.setVisible(true);
3953
3954
3955                // Display a dialog to the user telling them how many songs were added.
3956                int iAddedCnt = 0;
3957                int iAfterCnt = treeMgr.getIntSongCount();
3958                if (iAfterCnt > iBeforeCnt)
3959                   iAddedCnt = iAfterCnt - iBeforeCnt;
3960
3961
3962                JOptionPane pane   = new JOptionPane(iAddedCnt + " songs were added to the Jukebox Tree.",
JOptionPane.INFORMATION_MESSAGE);
3963                JDialog infoDialog = pane.createDialog(this, "Information");
3964                infoDialog.show();
3965             }
3966
3967
3968             // If we have just added new mp3s to an empty jukebox, then enable funtionality
3969             if ((tree.getRowCount() > 1) && (bIsAppFunctional == false))
3970             {
3971                bIsAppFunctional = true;
3972                enableFunctionality();
3973             }
3974          }
3975
3976          trace("menuAddDir_doWork()", EXIT);
3977       }
3978
3979       private void initControl(JPanel panel, JButton btn, int x1, int y1, int x2, int y2, boolean bEnable)
3980       {
3981          trace("initControl()", ENTER);
3982
3983          //btn.setBackground(Color.white);
3984          btn.setForeground(Color.blue);
3985          btn.setFont(new Font("SansSerif", Font.BOLD, 12));
3986          btn.setBounds(x1,y1,x2,y2);
3987          btn.setEnabled(bEnable);
3988          btn.addActionListener(this);
3989          panel.add(btn);
3990
3991          trace("initControl()", EXIT);
3992       }
3993
3994       private ImageIcon loadIcon(String name) throws java.lang.NullPointerException
3995       {
3996          //trace("loadIcon()", ENTER);
3997
3998          Object icon;
3999          String jarName = null;
4000          icon = new ImageIcon(name);
4001          if (((ImageIcon)icon).getIconWidth() == -1)
4002          {
4003             jarName = new String("/");
4004             jarName = jarName.concat(name);
4005
4006             try
4007             {
4008                icon = new ImageIcon(this.getClass().getResource(jarName));
4009             }
4010             catch (java.lang.NullPointerException e)
4011             {
4012                System.out.println(" ");
4013                System.out.println(" ");
4014                System.out.println("ERROR:  Could not find: " + name);
4015                System.out.println(" ");
4016                System.out.println(" ");
4017
4018                throw e;
4019             }
4020
4021             jarName = null;
4022          }
4023
4024          //trace("loadIcon()", EXIT);
4025
4026          return (ImageIcon)icon;
4027       }
4028
4029       private void updateVisibleCDTextField()
4030       {
4031          trace("updateVisibleCDTextField()", ENTER);
4032
4033          String strFirst = "";
4034          String strLast  = "";
4035
4036          switch (iVisiblePanel)
4037          {
4038             case TABLE_PANEL:
4039                if (tableCDPanel != null)
```

```
4040              {
4041                 visibleCDsTxtField.setText(tableCDPanel.getCDNumber());
4042              }
4043              break;
4044
4045           case SEARCH_PANEL:
4046              if (searchCDPanel != null)
4047              {
4048                 visibleCDsTxtField.setText(searchCDPanel.getCDNumber());
4049              }
4050              break;
4051
4052           case GENRE_PANEL:
4053              if (genreNorthCD != null)
4054              {
4055                 strFirst = genreNorthCD.getCDNumber();
4056                 strLast  = null;
4057                 if (genreSouthCD != null)
4058                 {
4059                    strLast = genreSouthCD.getCDNumber();
4060                    visibleCDsTxtField.setText(strFirst + " - " + strLast);
4061                 }
4062                 else
4063                 {
4064                    visibleCDsTxtField.setText(strFirst);
4065                 }
4066              }
4067              break;
4068
4069           default:
4070              if (northwestCD != null)
4071              {
4072                 strFirst = northwestCD.getCDNumber();
4073                 strLast = strFirst;
4074              }
4075
4076              if (northeastCD != null)
4077                 strLast = northeastCD.getCDNumber();
4078
4079              if (southwestCD != null)
4080                 strLast = southwestCD.getCDNumber();
4081
4082              if (southeastCD != null)
4083                 strLast = southeastCD.getCDNumber();
4084
4085              visibleCDsTxtField.setText(strFirst + " - " + strLast);
4086              break;
4087        }
4088
4089        trace("updateVisibleCDTextField()", EXIT);
4090     }
4091
4092     public void addToClassicPanel()
4093     {
4094        trace("addToClassicPanel()", ENTER);
4095
4096        if (northwestCD != null)
4097           classicPanel.add(northwestCD);
4098
4099        if (northeastCD != null)
4100           classicPanel.add(northeastCD);
4101
4102        if (southwestCD != null)
4103           classicPanel.add(southwestCD);
4104
4105        if (southeastCD != null)
4106           classicPanel.add(southeastCD);
4107
4108        updateVisibleCDTextField();
4109
4110        trace("addToClassicPanel()", EXIT);
4111     }
4112
4113     public void clearSelections()
4114     {
4115        trace("clearSelections()", ENTER);
4116
4117        if (northwestCD != null)
4118           northwestCD.clearSelection();
4119
4120        if (northeastCD != null)
4121           northeastCD.clearSelection();
4122
4123        if (southwestCD != null)
4124           southwestCD.clearSelection();
4125
4126        if (southeastCD != null)
4127           southeastCD.clearSelection();
4128
4129        trace("clearSelections()", EXIT);
```

```
4130      }
4131
4132      public void actionPerformed(java.awt.event.ActionEvent event)
4133      {
4134         //trace("actionPerformed()", ENTER);
4135
4136         Object object = event.getSource();
4137
4138
4139         if (object != timer)
4140         {
4141            iElapsedUserInactivity = 0;
4142         }
4143         else
4144         {
4145            iElapsedUserInactivity = iElapsedUserInactivity + 1;
4146         }
4147
4148
4149         if (object == timer)
4150         {
4151            timer_actionPerformed();
4152         }
4153         else if (object == srchBtn || object == searchCancel)
4154         {
4155            if (object == srchBtn)
4156               trace("actionPerformed::srchBtn", ENTER);
4157            else
4158               trace("actionPerformed::searchCancel", ENTER);
4159
4160
4161            trace("iVisiblePanel= " + iVisiblePanel);
4162
4163
4164            if (iVisiblePanel == SEARCH_PANEL)
4165            {
4166               srchBtn.setEnabled(true);
4167
4168               if (searchCDPanel != null)
4169               {
4170                  searchPanel.remove(searchCDPanel);
4171                  searchCDPanel = null;
4172               }
4173
4174               iSearchMp3Row = -1;
4175               searchText = "";
4176               searchTextField.setText("");
4177
4178               searchVector = null;
4179               searchVector = new Vector();
4180
4181
4182               searchPanel.setVisible(false);
4183               searchScroll.getViewport().remove(searchTable);
4184
4185               searchTableVector = createSearchTableVector(searchVector);
4186               searchTableModel = new DefaultTableModel(searchVector, searchColumnHeaderVect);
4187               searchTable = createSearchTable(searchTableModel);
4188
4189               searchPanel.setVisible(true);
4190               searchScroll.getViewport().add(searchTable);
4191
4192
4193
4194               topBtn.setEnabled(true);
4195               nextGenreBtn.setEnabled(true);
4196               prevGenreBtn.setEnabled(true);
4197               btmBtn.setEnabled(true);
4198               prevPageBtn.setEnabled(true);
4199               nextPageBtn.setEnabled(true);
4200
4201               if (playerMgr.getStatus() == PlayerMgr.STOPPED)
4202                  showCurrentBtn.setEnabled(false);
4203               else
4204                  showCurrentBtn.setEnabled(true);
4205
4206
4207               genreBtn.setEnabled(true);
4208               srchBtn.setEnabled(true);
4209               tableBtn.setEnabled(true);
4210
4211
4212               checkScrollButtons();
4213
4214               iLastVisiblePanel = iVisiblePanel;
4215               iVisiblePanel = CLASSIC_PANEL;
4216
4217               updateVisibleCDTextField();
4218
4219               userCardMgr.show(userPanel, "classic");
```

```
4220            }
4221            else
4222            {
4223               checkSearchScrollButtons();
4224
4225               visibleCDsTxtField.setText("");
4226
4227               srchBtn.setEnabled(false);
4228
4229               genreBtn.setEnabled(true);
4230               tableBtn.setEnabled(true);
4231
4232               topBtn.setEnabled(false);
4233               nextGenreBtn.setEnabled(false);
4234               prevGenreBtn.setEnabled(false);
4235               btmBtn.setEnabled(false);
4236               prevPageBtn.setEnabled(false);
4237               nextPageBtn.setEnabled(false);
4238               showCurrentBtn.setEnabled(false);
4239
4240               userCardMgr.show(userPanel, "search");
4241               userCardMgr.show(userPanel, "classic");
4242
4243               iLastVisiblePanel = iVisiblePanel;
4244               iVisiblePanel = SEARCH_PANEL;
4245               userCardMgr.show(userPanel, "search");
4246            }
4247
4248            if (object == srchBtn)
4249               trace("actionPerformed::srchBtn", EXIT);
4250            else
4251               trace("actionPerformed::searchCancel", EXIT);
4252         }
4253         else if (object == searchPageUpBtn)
4254         {
4255            trace("actionPerformed::searchPageUpBtn", ENTER);
4256
4257            int i = searchTable.getSelectionModel().getLeadSelectionIndex();
4258
4259            if (i > 0)
4260            {
4261               searchTable.getSelectionModel().setLeadSelectionIndex(i - 1);
4262               searchTable.changeSelection(i - 1 , 1, false, false);
4263            }
4264
4265            checkSearchScrollButtons();
4266
4267            trace("actionPerformed::searchPageUpBtn", EXIT);
4268         }
4269         else if (object == searchPageDnBtn)
4270         {
4271            trace("actionPerformed::searchPageDnBtn", ENTER);
4272
4273            int i = searchTable.getSelectionModel().getLeadSelectionIndex();
4274
4275            if (i < searchVector.size() - 1)
4276            {
4277               searchTable.getSelectionModel().setLeadSelectionIndex(i + 1);
4278               searchTable.changeSelection(i + 1, 1, false, false);
4279            }
4280
4281            checkSearchScrollButtons();
4282
4283            trace("actionPerformed::searchPageDnBtn", EXIT);
4284         }
4285         else if (object == searchByArtistBtn)
4286         {
4287            trace("actionPerformed::searchByArtistBtn", ENTER);
4288
4289            iSearchBy = TreeMgr.BY_ARTIST;
4290
4291            searchByArtistBtn.setEnabled(false);
4292            searchBySongBtn.setEnabled(true);
4293            searchByCDTitleBtn.setEnabled(true);
4294            searchByAllBtn.setEnabled(true);
4295
4296            trace("actionPerformed::searchByArtistBtn", EXIT);
4297         }
4298         else if (object == searchBySongBtn)
4299         {
4300            trace("actionPerformed::searchBySongBtn", ENTER);
4301
4302            iSearchBy = TreeMgr.BY_SONG;
4303
4304            searchByArtistBtn.setEnabled(true);
4305            searchBySongBtn.setEnabled(false);
4306            searchByCDTitleBtn.setEnabled(true);
4307            searchByAllBtn.setEnabled(true);
4308
4309            trace("actionPerformed::searchBySongBtn", EXIT);
```

```
4310        }
4311        else if (object == searchByCDTitleBtn)
4312        {
4313           trace("actionPerformed::searchByCDTitleBtn", ENTER);
4314
4315           iSearchBy = TreeMgr.BY_CDTITLE;
4316
4317           searchByArtistBtn.setEnabled(true);
4318           searchBySongBtn.setEnabled(true);
4319           searchByCDTitleBtn.setEnabled(false);
4320           searchByAllBtn.setEnabled(true);
4321
4322           trace("actionPerformed::searchByCDTitle", EXIT);
4323        }
4324        else if (object == searchByAllBtn)
4325        {
4326           trace("actionPerformed::searchByAllBtn", ENTER);
4327
4328           iSearchBy = TreeMgr.BY_ALL;
4329
4330           searchByArtistBtn.setEnabled(true);
4331           searchBySongBtn.setEnabled(true);
4332           searchByCDTitleBtn.setEnabled(true);
4333           searchByAllBtn.setEnabled(false);
4334
4335           trace("actionPerformed::searchByAllBtn", EXIT);
4336        }
4337        else if (object == tablePageUpBtn)
4338        {
4339           trace("actionPerformed::tablePageUpBtn", ENTER);
4340
4341           int i = table.getSelectionModel().getLeadSelectionIndex();
4342
4343           if (i > 0)
4344           {
4345              table.getSelectionModel().setLeadSelectionIndex(i - 1);
4346              table.changeSelection(i - 1 , 1, false, false);
4347           }
4348
4349           checkTableScrollButtons();
4350
4351           trace("actionPerformed::tablePageUpBtn", EXIT);
4352        }
4353        else if (object == tablePageDnBtn)
4354        {
4355           trace("actionPerformed::tablePageDnBtn", ENTER);
4356
4357           int i = table.getSelectionModel().getLeadSelectionIndex();
4358
4359           if (i < tableVector.size() - 1)
4360           {
4361              table.getSelectionModel().setLeadSelectionIndex(i + 1);
4362              table.changeSelection(i + 1, 1, false, false);
4363           }
4364
4365           checkTableScrollButtons();
4366
4367           trace("actionPerformed::tablePageDnBtn", EXIT);
4368        }
4369        else if (object == searchSearch)
4370        {
4371           trace("actionPerformed::searchSearch", ENTER);
4372
4373           state = true;
4374           searchText = searchTextField.getText();
4375           searchTextField.setText("");
4376
4377           if (searchText.length() >= 3 && !searchText.equals("   "))
4378           {
4379              searchVector = null;
4380              searchVector = new Vector();
4381
4382              searchPanel.setVisible(false);
4383              searchScroll.getViewport().remove(searchTable);
4384
4385              searchTableVector = createSearchTableVector(searchVector);
4386              searchTableModel = new DefaultTableModel(searchVector, searchColumnHeaderVect);
4387              searchTable = createSearchTable(searchTableModel);
4388
4389              searchPanel.setVisible(true);
4390              searchScroll.getViewport().add(searchTable);
4391
4392
4393              searchVector = treeMgr.search(searchText, iSearchBy, TreeMgr.BOOLEAN_AND, 100);
4394
4395
4396              if (searchVector == null)
4397              {
4398                 if (searchCDPanel != null)
4399                 {
```

```
4400                    searchPanel.remove(searchCDPanel);
4401                    searchCDPanel = null;
4402                 }
4403
4404                 JOptionPane pane   = new JOptionPane("Error:  Could not find any artists or songs using: '"+
searchText + "'.  Please try again.", JOptionPane.ERROR_MESSAGE);
4405                 JDialog infoDialog = pane.createDialog(this, "Information");
4406                 infoDialog.show();
4407              }
4408              else
4409              {
4410                 searchPanel.setVisible(false);
4411                 searchScroll.getViewport().remove(searchTable);
4412
4413                 searchTableVector = createSearchTableVector(searchVector);
4414                 searchTableModel = new DefaultTableModel(searchTableVector, searchColumnHeaderVect);
4415                 searchTable = createSearchTable(searchTableModel);
4416
4417                 searchScroll.getViewport().add(searchTable);
4418
4419                 if (searchVector.size() > 0)
4420                 {
4421                    searchTable.getSelectionModel().setLeadSelectionIndex(0);
4422
4423                    searchTable.changeSelection(0, 1, false, false);
4424                 }
4425                 else
4426                 {
4427                    if (searchCDPanel != null)
4428                    {
4429                       searchPanel.remove(searchCDPanel);
4430                       searchCDPanel = null;
4431                    }
4432                 }
4433
4434                 searchPanel.setVisible(true);
4435              }
4436
4437              checkSearchScrollButtons();
4438           }
4439           else
4440           {
4441              JOptionPane pane = new JOptionPane("ERROR:  The Search field must contain at least 3 non-blank
characters...  Please try again.", JOptionPane.ERROR_MESSAGE);
4442              JDialog infoDialog = pane.createDialog(this, "Information");
4443              infoDialog.show();
4444           }
4445
4446           trace("actionPerformed::searchSearch", EXIT);
4447        }
4448        else if (object == searchClear)
4449        {
4450           trace("actionPerformed::searchClear", ENTER);
4451
4452           if (searchCDPanel != null)
4453           {
4454              searchPanel.remove(searchCDPanel);
4455              searchCDPanel = null;
4456           }
4457
4458           iSearchMp3Row = -1;
4459           searchText = "";
4460           searchTextField.setText("");
4461
4462           searchVector = null;
4463           searchVector = new Vector();
4464
4465           searchPanel.setVisible(false);
4466           searchScroll.getViewport().remove(searchTable);
4467
4468           searchTableVector = createSearchTableVector(searchVector);
4469           searchTableModel = new DefaultTableModel(searchVector, searchColumnHeaderVect);
4470           searchTable = createSearchTable(searchTableModel);
4471
4472           searchPanel.setVisible(true);
4473           searchScroll.getViewport().add(searchTable);
4474
4475           checkSearchScrollButtons();
4476
4477           trace("actionPerformed::searchClear", EXIT);
4478        }
4479        else if (object == showCurrentBtn)
4480        {
4481           trace("actionPerformed::showCurrentBtn", ENTER);
4482
4483           String strCurrentSong = treeMgr.getCurrentlyPlayingSong();
4484           trace("Current: " + strCurrentSong);
4485
4486           treeMgr.setSelectedMp3BySong(strCurrentSong, tree);
4487           PlayListEntry mp3 = playerMgr.getCurrentPlayListObject();
```

```
4488
4489
4490            // Make the corresponding selection for the main window.
4491            treeMgr.setSelectedCDParentRowBySong(tree, mp3);
4492            iCurrentCDPtr = getCDPtrForSelectedCD();
4493            iCurrentGenrePtr = getGenrePtrForSelectedCD();
4494
4495
4496            userPanel.setVisible(false);
4497            initClassicPanel(CURRENT);
4498            addToClassicPanel();
4499
4500
4501            // Then, make the song the selection in its CD Panel (but deselect the selection txt field).
4502            northwestCD.setSelectedSong(mp3);
4503            selectionTxtField.setText("");
4504            checkScrollButtons();
4505
4506
4507            userPanel.setVisible(true);
4508
4509            trace("actionPerformed::showCurrentBtn", EXIT);
4510         }
4511         else if (object == genreBtn || object == genreCloseBtn)
4512         {
4513            if (object == genreBtn)
4514               trace("actionPerformed::genreBtn", ENTER);
4515            else
4516               trace("actionPerformed::genreCloseBtn", ENTER);
4517
4518            userPanel.setVisible(false);
4519
4520            if (iVisiblePanel == GENRE_PANEL)
4521            {
4522               topBtn.setEnabled(true);
4523               nextGenreBtn.setEnabled(true);
4524               prevGenreBtn.setEnabled(true);
4525               btmBtn.setEnabled(true);
4526               prevPageBtn.setEnabled(true);
4527               nextPageBtn.setEnabled(true);
4528
4529               if (playerMgr.getStatus() == PlayerMgr.STOPPED)
4530                  showCurrentBtn.setEnabled(false);
4531               else
4532                  showCurrentBtn.setEnabled(true);
4533
4534               genreBtn.setEnabled(true);
4535               srchBtn.setEnabled(true);
4536               tableBtn.setEnabled(true);
4537
4538               checkScrollButtons();
4539
4540               iLastVisiblePanel = iVisiblePanel;
4541               iVisiblePanel = CLASSIC_PANEL;
4542
4543               updateVisibleCDTextField();
4544
4545               userCardMgr.show(userPanel, "classic");
4546            }
4547            else
4548            {
4549               srchBtn.setEnabled(true);
4550               tableBtn.setEnabled(true);
4551
4552               genreBtn.setEnabled(false);
4553
4554               topBtn.setEnabled(false);
4555               nextGenreBtn.setEnabled(false);
4556               prevGenreBtn.setEnabled(false);
4557               btmBtn.setEnabled(false);
4558               prevPageBtn.setEnabled(false);
4559               nextPageBtn.setEnabled(false);
4560               showCurrentBtn.setEnabled(false);
4561
4562
4563               userCardMgr.show(userPanel, "genre");
4564               userCardMgr.show(userPanel, "classic");
4565
4566               iLastVisiblePanel = iVisiblePanel;
4567               iVisiblePanel = GENRE_PANEL;
4568               userCardMgr.show(userPanel, "genre");
4569
4570               int iCDPtr = iCurrentCDPtr;
4571
4572               // Select the corresponding genre in the listbox.
4573               genreList.setSelectedIndex(iCurrentGenrePtr);
4574
4575               // Advance the CDPanels in the Genre Panel if necessary.
4576               addToGenrePanel(iCDPtr);
4577            }
```

```
4578
4579            userPanel.setVisible(true);
4580
4581            if (object == genreBtn)
4582               trace("actionPerformed::genreBtn", EXIT);
4583            else
4584               trace("actionPerformed::genreCloseBtn", EXIT);
4585         }
4586         else if (object == genrePageUpBtn)
4587         {
4588            trace("actionPerformed::genrePageUpBtn", ENTER);
4589
4590            // See if we are at the bottom of the tree.
4591            int iCDPtr = iCurrentCDPtr;
4592            boolean bTop = false;
4593
4594            if (iCDPtr - 2 >= 0)
4595            {
4596               iCDPtr = iCDPtr - 2;
4597            }
4598            else if (iCDPtr -1 >= 0)
4599            {
4600               iCDPtr = iCDPtr - 1;
4601            }
4602            else
4603            {
4604               bTop = true;
4605            }
4606
4607
4608            // If we passed a genre boundary, then roll down to the bottom of the current genre.
4609            setSelectedCDForCDPtr(iCDPtr);
4610            int iGenrePtr = getGenrePtrForSelectedCD();
4611
4612            boolean bBottom = false;
4613            if (iGenrePtr != iCurrentGenrePtr)
4614            {
4615               if (iCurrentGenrePtr == iMaxGenrePtr)
4616               {
4617                  bBottom = true;
4618                  iCDPtr = iMaxCDPtr - 1;
4619               }
4620            }
4621
4622            if ( (bBottom == false && iGenrePtr != iCurrentGenrePtr) || bTop == true)
4623            {
4624               int iCDRow = ((Integer)CDVector.elementAt(iCDPtr)).intValue();
4625               int iGenreRow = ((Integer)GenreVector.elementAt(iCurrentGenrePtr + 1)).intValue();
4626
4627               while (iCDRow < iGenreRow)
4628               {
4629                  iCDPtr = iCDPtr + 1;
4630                  iCDRow = ((Integer)CDVector.elementAt(iCDPtr)).intValue();
4631
4632                  if (iCDRow > iGenreRow)
4633                  {
4634                     iCurrentCDPtr = iCDPtr - 2;
4635                  }
4636               }
4637            }
4638            else
4639            {
4640               iCurrentCDPtr = iCDPtr;
4641            }
4642
4643
4644            // Select the CD in the tree.
4645            setSelectedCDForCDPtr(iCurrentCDPtr);
4646
4647
4648            // Select the corresponding genre in the listbox.
4649            genreList.setSelectedIndex(iCurrentGenrePtr);
4650
4651
4652            // Advance the CDPanels in the Genre Panel if necessary.
4653            addToGenrePanel(iCurrentCDPtr);
4654
4655            trace("actionPerformed::genrePageUpBtn", EXIT);
4656         }
4657         else if (object == genrePageDnBtn)
4658         {
4659            trace("actionPerformed::genrePageDnBtn", ENTER);
4660
4661            // See if we are at the bottom of the tree.
4662            int iCDPtr = iCurrentCDPtr;
4663            boolean bBottom = false;
4664
4665            if (iCDPtr + 2 <= iMaxCDPtr)
4666            {
4667               iCDPtr = iCDPtr + 2;
```

```
4668            }
4669            else
4670            {
4671               bBottom = true;
4672            }
4673
4674
4675            // If we passed a genre boundary, then roll up to the top of the current genre.
4676            setSelectedCDForCDPtr(iCDPtr);
4677            int iGenrePtr = getGenrePtrForSelectedCD();
4678
4679            boolean bTop = false;
4680            if (iGenrePtr != iCurrentGenrePtr)
4681            {
4682               if (iCurrentGenrePtr == 0)
4683               {
4684                  bTop = true;
4685                  iCDPtr = 0;
4686               }
4687            }
4688
4689            if ( (bTop == false && iGenrePtr != iCurrentGenrePtr) || bBottom == true)
4690            {
4691               int iCDRow = ((Integer)CDVector.elementAt(iCDPtr)).intValue();
4692               int iGenreRow = ((Integer)GenreVector.elementAt(iCurrentGenrePtr)).intValue();
4693
4694               while (iCDRow > iGenreRow)
4695               {
4696                  iCDPtr = iCDPtr - 1;
4697                  iCDRow = ((Integer)CDVector.elementAt(iCDPtr)).intValue();
4698
4699                  if (iCDRow < iGenreRow)
4700                  {
4701                     iCurrentCDPtr = iCDPtr + 1;
4702                  }
4703               }
4704            }
4705            else
4706            {
4707               iCurrentCDPtr = iCDPtr;
4708            }
4709
4710
4711            // Select the CD in the tree.
4712            setSelectedCDForCDPtr(iCurrentCDPtr);
4713
4714
4715            // Select the corresponding genre in the listbox.
4716            genreList.setSelectedIndex(iCurrentGenrePtr);
4717
4718
4719            // Advance the CDPanels in the Genre Panel if necessary.
4720            addToGenrePanel(iCurrentCDPtr);
4721
4722            trace("actionPerformed::genrePageDnBtn", EXIT);
4723         }
4724         else if (object == tableBtn || object == tableCloseBtn)
4725         {
4726            if (object == tableBtn)
4727               trace("actionPerformed::tableBtn", ENTER);
4728            else
4729               trace("actionPerformed::tableCloseBtn", ENTER);
4730
4731            userPanel.setVisible(false);
4732
4733            if (iVisiblePanel == TABLE_PANEL)
4734            {
4735               tableBtn.setEnabled(true);
4736
4737               topBtn.setEnabled(true);
4738               nextGenreBtn.setEnabled(true);
4739               prevGenreBtn.setEnabled(true);
4740               btmBtn.setEnabled(true);
4741               prevPageBtn.setEnabled(true);
4742               nextPageBtn.setEnabled(true);
4743
4744               if (playerMgr.getStatus() == PlayerMgr.STOPPED)
4745                  showCurrentBtn.setEnabled(false);
4746               else
4747                  showCurrentBtn.setEnabled(true);
4748
4749               selectionTxtField.setText("");
4750
4751               checkScrollButtons();
4752
4753               iLastVisiblePanel = iVisiblePanel;
4754               iVisiblePanel = CLASSIC_PANEL;
4755
4756               updateVisibleCDTextField();
4757
```

```
4758            userCardMgr.show(userPanel, "classic");
4759         }
4760         else
4761         {
4762            srchBtn.setEnabled(true);
4763            genreBtn.setEnabled(true);
4764
4765            tableBtn.setEnabled(false);
4766
4767            userCardMgr.show(userPanel, "table");
4768            userCardMgr.show(userPanel, "classic");
4769
4770            iLastVisiblePanel = iVisiblePanel;
4771            iVisiblePanel = TABLE_PANEL;
4772            userCardMgr.show(userPanel, "table");
4773
4774            userPanel.setVisible(true);
4775
4776            table.getSelectionModel().clearSelection();
4777            table.getSelectionModel().setLeadSelectionIndex(0);
4778            table.changeSelection(0, 1, false, false);
4779
4780            checkScrollButtons();
4781            updateVisibleCDTextField();
4782
4783            topBtn.setEnabled(false);
4784            nextGenreBtn.setEnabled(false);
4785            prevGenreBtn.setEnabled(false);
4786            btmBtn.setEnabled(false);
4787            prevPageBtn.setEnabled(false);
4788            nextPageBtn.setEnabled(false);
4789            showCurrentBtn.setEnabled(false);
4790         }
4791
4792         checkTableScrollButtons();
4793
4794         userPanel.setVisible(true);
4795
4796         if (object == tableBtn)
4797            trace("actionPerformed::tableBtn", EXIT);
4798         else
4799            trace("actionPerformed::tableCloseBtn", EXIT);
4800      }
4801      else if (object == treeViewBtn)
4802      {
4803         trace("actionPerformed::treeViewBtn", ENTER);
4804
4805         if (iVisiblePanel != ADMIN_PANEL)
4806         {
4807            logInfo("User entered the logon screen.");
4808
4809            // Bring up the logon dialog.
4810            JFrame logonFrame = new JFrame();
4811            LogonDialog logonDialog = new LogonDialog(logonFrame, "Logon", true);
4812
4813            if (logonDialog.getState())
4814            {
4815               String userid = logonDialog.getUserid();
4816               logInfo("UserID=" + userid);
4817
4818               if (logonDialog.getMode().equals("OWNER"))
4819               {
4820                  logInfo("Owner mode.");
4821
4822                  bOwnerMode = true;
4823                  visibleCDsTxtField.setText("");
4824                  disableBottomPanel();
4825
4826                  ownerIncrementBtn.setVisible(true);
4827                  ownerDecrementBtn.setVisible(true);
4828                  ownerAddPathBtn.setVisible(true);
4829                  ownerDeleteFromDiskBtn.setVisible(true);
4830                  ownerAddNodeToQBtn.setVisible(true);
4831                  ownerNumToQueueSB.setVisible(true);
4832                  ownerRemNodeFromQBtn.setVisible(true);
4833                  ownerResetTreeBtn.setVisible(true);
4834
4835                  iLastVisiblePanel = iVisiblePanel;
4836                  iVisiblePanel = ADMIN_PANEL;
4837
4838                  loadLogFile();
4839
4840                  cardPanel.setVisible(false);
4841
4842                  cardMgr.show(cardPanel, "admin");
4843                  cardPanel.setVisible(true);
4844               }
4845               else if (logonDialog.getMode().equals("ADMIN"))
4846               {
4847                  logInfo("Admin mode.");
```

```
                    bOwnerMode = false;
                    visibleCDsTxtField.setText("");
                    disableBottomPanel();

                    ownerIncrementBtn.setVisible(false);
                    ownerDecrementBtn.setVisible(false);
                    ownerAddPathBtn.setVisible(false);
                    ownerDeleteFromDiskBtn.setVisible(false);
                    ownerAddNodeToQBtn.setVisible(false);
                    ownerNumToQueueSB.setVisible(false);
                    ownerRemNodeFromQBtn.setVisible(false);
                    ownerResetTreeBtn.setVisible(false);

                    iLastVisiblePanel = iVisiblePanel;
                    iVisiblePanel = ADMIN_PANEL;

                    cardPanel.setVisible(false);
                    cardMgr.show(cardPanel, "admin");
                    cardPanel.setVisible(true);
                }
                else
                {
                    logInfo("Invalid Logon Attempt.");

                    bOwnerMode = false;
                    checkBottomPanel();

                    ownerIncrementBtn.setVisible(false);
                    ownerDecrementBtn.setVisible(false);
                    ownerAddPathBtn.setVisible(false);
                    ownerDeleteFromDiskBtn.setVisible(false);
                    ownerAddNodeToQBtn.setVisible(false);
                    ownerNumToQueueSB.setVisible(false);
                }
            }
        }
        else
        {
            bOwnerMode = false;

            iLastVisiblePanel = ADMIN_PANEL;
            iVisiblePanel = iLastVisiblePanel;
            checkBottomPanel();

            cardPanel.setVisible(false);
            cardMgr.show(cardPanel, "user");
            cardPanel.setVisible(true);
        }

        trace("actionPerformed::treeViewBtn", EXIT);
    }
    else if (object == nextPageBtn)
    {
        trace("actionPerformed::nextPageBtn", ENTER);

        cardPanel.setVisible(false);
        cardMgr.next(cardPanel);

        nextPageBtn.setEnabled(false);
        prevPageBtn.setEnabled(true);

        iCurrentCDPtr = iCurrentCDPtr + initClassicPanel(NEXT);

        int iGenreRow = 0;
        int iCDRow = ((Integer)CDVector.elementAt(iCurrentCDPtr)).intValue();
        boolean bDone = false;

        while (!bDone)
        {
            if (iCurrentGenrePtr + 1 <= iMaxGenrePtr)
            {
                iGenreRow = ((Integer)GenreVector.elementAt(iCurrentGenrePtr + 1)).intValue();

                if (iGenreRow < iCDRow)
                    iCurrentGenrePtr = iCurrentGenrePtr + 1;
                else
                    bDone = true;
            }
            else
                bDone = true;
        }

        addToClassicPanel();
        checkScrollButtons();

        classicPanel.repaint(classicPanel.getVisibleRect());

        cardMgr.show(cardPanel, "user");
        cardPanel.setVisible(true);
```

```
4939            trace("actionPerformed::nextPageBtn", EXIT);
4940         }
4941         else if (object == prevPageBtn)
4942         {
4943            trace("actionPerformed::prevPageBtn", ENTER);
4944
4945            cardPanel.setVisible(false);
4946            cardMgr.next(cardPanel);
4947
4948            prevPageBtn.setEnabled(false);
4949            nextPageBtn.setEnabled(true);
4950
4951            iCurrentCDPtr = iCurrentCDPtr - initClassicPanel(PREVIOUS);
4952
4953            int iGenreRow = 0;
4954            int iCDRow = ((Integer)CDVector.elementAt(iCurrentCDPtr)).intValue();
4955
4956            boolean bDone = false;
4957            while (!bDone)
4958            {
4959               if (iCurrentGenrePtr > 0)
4960               {
4961                  iGenreRow = ((Integer)GenreVector.elementAt(iCurrentGenrePtr)).intValue();
4962
4963                  if (iCDRow < iGenreRow)
4964                     iCurrentGenrePtr = iCurrentGenrePtr - 1;
4965                  else
4966                     bDone = true;
4967               }
4968               else
4969                  bDone = true;
4970            }
4971
4972            addToClassicPanel();
4973            checkScrollButtons();
4974
4975            classicPanel.repaint(classicPanel.getVisibleRect());
4976
4977            cardMgr.show(cardPanel, "user");
4978            cardPanel.setVisible(true);
4979
4980            trace("actionPerformed::prevPageBtn", EXIT);
4981         }
4982         else if (object == topBtn)
4983         {
4984            trace("actionPerformed::topBtn", ENTER);
4985
4986            cardPanel.setVisible(false);
4987            cardMgr.next(cardPanel);
4988
4989            iCurrentCDPtr = 0;
4990            iCurrentGenrePtr = 0;
4991            initClassicPanel(CURRENT);
4992
4993            addToClassicPanel();
4994            checkScrollButtons();
4995
4996            classicPanel.repaint(classicPanel.getVisibleRect());
4997
4998            cardMgr.show(cardPanel, "user");
4999            cardPanel.setVisible(true);
5000
5001            trace("actionPerformed::topBtn", EXIT);
5002         }
5003         else if (object == btmBtn)
5004         {
5005            trace("actionPerformed::btmBtn", ENTER);
5006
5007            cardPanel.setVisible(false);
5008            cardMgr.next(cardPanel);
5009
5010            if (iMaxCDPtr - 3 >= 0)
5011               iCurrentCDPtr = iMaxCDPtr - 3;
5012            else if (iMaxCDPtr - 2 >= 0)
5013               iCurrentCDPtr = iMaxCDPtr - 2;
5014            else if (iMaxCDPtr - 1 >= 0)
5015               iCurrentCDPtr = iMaxCDPtr - 1;
5016            else
5017               iCurrentCDPtr = iMaxCDPtr;
5018
5019            iCurrentGenrePtr = iMaxGenrePtr;
5020            initClassicPanel(CURRENT);
5021
5022            addToClassicPanel();
5023            checkScrollButtons();
5024
5025            classicPanel.repaint(classicPanel.getVisibleRect());
5026
5027            cardMgr.show(cardPanel, "user");
5028            cardPanel.setVisible(true);
```

```
5029
5030            trace("actionPerformed::btmBtn", EXIT);
5031         }
5032         else if (object == prevGenreBtn)
5033         {
5034            trace("actionPerformed::prevGenreBtn", ENTER);
5035
5036            cardPanel.setVisible(false);
5037            cardMgr.next(cardPanel);
5038
5039            int iTmpCDPtr = iCurrentCDPtr;
5040            int iGenreRow = 0;
5041
5042            if (iCurrentGenrePtr > 0)
5043            {
5044               iCurrentGenrePtr = iCurrentGenrePtr - 1;
5045               iGenreRow = ((Integer)GenreVector.elementAt(iCurrentGenrePtr)).intValue();
5046            }
5047            else
5048            {
5049               iCurrentGenrePtr = 0;
5050               iGenreRow = ((Integer)GenreVector.elementAt(iCurrentGenrePtr)).intValue();
5051            }
5052
5053            int iCDRow = ((Integer)CDVector.elementAt(iTmpCDPtr)).intValue();
5054
5055            boolean bDone = false;
5056            while (!bDone)
5057            {
5058               if (iTmpCDPtr > 0)
5059               {
5060                  iCDRow = ((Integer)CDVector.elementAt(iTmpCDPtr - 1)).intValue();
5061
5062                  if (iCDRow > iGenreRow)
5063                  {
5064                     iTmpCDPtr = iTmpCDPtr - 1;
5065                  }
5066                  else
5067                     bDone = true;
5068               }
5069               else
5070                  bDone = true;
5071            }
5072            iCurrentCDPtr = iTmpCDPtr;
5073
5074            initClassicPanel(CURRENT);
5075
5076            addToClassicPanel();
5077            checkScrollButtons();
5078
5079            classicPanel.repaint(classicPanel.getVisibleRect());
5080
5081            cardMgr.show(cardPanel, "user");
5082            cardPanel.setVisible(true);
5083
5084            trace("actionPerformed::prevGenreBtn", EXIT);
5085         }
5086         else if (object == nextGenreBtn)
5087         {
5088            trace("actionPerformed::nextGenreBtn", ENTER);
5089
5090            cardPanel.setVisible(false);
5091            cardMgr.next(cardPanel);
5092
5093            int iTmpCDPtr = iCurrentCDPtr;
5094            int iGenreRow = 0;
5095
5096            if (iCurrentGenrePtr < iMaxGenrePtr)
5097            {
5098               iCurrentGenrePtr = iCurrentGenrePtr + 1;
5099               iGenreRow = ((Integer)GenreVector.elementAt(iCurrentGenrePtr)).intValue();
5100            }
5101            else
5102            {
5103               iCurrentGenrePtr = iMaxGenrePtr;
5104               iGenreRow = ((Integer)GenreVector.elementAt(iMaxGenrePtr)).intValue();
5105            }
5106
5107            int iCDRow = ((Integer)CDVector.elementAt(iTmpCDPtr)).intValue();
5108
5109            while (iCDRow < iGenreRow)
5110            {
5111               iTmpCDPtr = iTmpCDPtr + 1;
5112               iCDRow = ((Integer)CDVector.elementAt(iTmpCDPtr)).intValue();
5113            }
5114            iCurrentCDPtr = iTmpCDPtr;
5115
5116            initClassicPanel(CURRENT);
5117
5118            addToClassicPanel();
```

```
5119            checkScrollButtons();
5120
5121            classicPanel.repaint(classicPanel.getVisibleRect());
5122
5123            cardMgr.show(cardPanel, "user");
5124            cardPanel.setVisible(true);
5125
5126            trace("actionPerformed::nextGenreBtn", EXIT);
5127         }
5128         else if (object == btn_1)
5129         {
5130            strText = strText + "1";
5131            selectionTxtField.setText(strText);
5132            checkButtons();
5133         }
5134         else if (object == btn_2)
5135         {
5136            strText = strText + "2";
5137            selectionTxtField.setText(strText);
5138            checkButtons();
5139         }
5140         else if (object == btn_3)
5141         {
5142            strText = strText + "3";
5143            selectionTxtField.setText(strText);
5144            checkButtons();
5145         }
5146         else if (object == btn_4)
5147         {
5148            strText = strText + "4";
5149            selectionTxtField.setText(strText);
5150            checkButtons();
5151         }
5152         else if (object == btn_5)
5153         {
5154            strText = strText + "5";
5155            selectionTxtField.setText(strText);
5156            checkButtons();
5157         }
5158         else if (object == btn_6)
5159         {
5160            strText = strText + "6";
5161            selectionTxtField.setText(strText);
5162            checkButtons();
5163         }
5164         else if (object == btn_7)
5165         {
5166            strText = strText + "7";
5167            selectionTxtField.setText(strText);
5168            checkButtons();
5169         }
5170         else if (object == btn_8)
5171         {
5172            strText = strText + "8";
5173            selectionTxtField.setText(strText);
5174            checkButtons();
5175         }
5176         else if (object == btn_9)
5177         {
5178            strText = strText + "9";
5179            selectionTxtField.setText(strText);
5180            checkButtons();
5181         }
5182         else if (object == btn_0)
5183         {
5184            strText = strText + "0";
5185            selectionTxtField.setText(strText);
5186            checkButtons();
5187         }
5188         else if (object == cancelBtn)
5189         {
5190            strText = null;
5191            strText = "";
5192            selectionTxtField.setText(strText);
5193            checkButtons();
5194         }
5195         else if (object == enterBtn)
5196         {
5197            trace("actionPerformed::enterBtn", ENTER);
5198
5199            clearSelections();
5200
5201            strText = selectionTxtField.getText();
5202
5203            PlayListEntry mp3 = treeMgr.getSongForJukeboxNo(tree, strText);
5204
5205            if (mp3 != null)
5206            {
5207               int iAnswer = ConfirmationDialog.NO_OPTION;
5208
```

```
5209                // If configured to confirm selection, then display dialog.  Otherwise, go ahead and add
5210                // the song to the playlist.
5211                if (bShowConfirmation == true)
5212                {
5213                   JFrame confirmationFrame = new JFrame();
5214                   String strSong = mp3.toString().substring(0, mp3.toString().indexOf(".mp3"));
5215                   ConfirmationDialog confirmationDialog = new ConfirmationDialog(confirmationFrame, "   Selection
Confirmation", "Do you wish to select the following song?: ", strSong);
5216                   strSong = null;
5217
5218                   iAnswer = confirmationDialog.getValue();
5219                }
5220                else
5221                {
5222                   iAnswer = ConfirmationDialog.YES_OPTION;
5223                }
5224
5225                repaintCDPanels();
5226
5227                if (iAnswer == ConfirmationDialog.YES_OPTION)
5228                {
5229                   playerMgr.addPaidSongToPlayList(mp3);
5230                   bDirtyFlag = true;
5231
5232                   playlistVector = playerMgr.getPlayListVector();
5233                   playlistList.setListData(playlistVector);
5234
5235                   playlistList.repaint(playlistList.getVisibleRect());
5236                   tree.repaint(tree.getVisibleRect());
5237
5238
5239                   credits = credits - 1;
5240                   newCredits = credits;
5241
5242                   checkButtons();
5243
5244                   intCredits = null;
5245                   intCredits = new Integer(credits);
5246
5247                   creditsTxtField.setText(intCredits.toString());
5248
5249                   logInfo("Adding to the playlist: " + mp3.toString() + " Credits=" + credits);
5250
5251                   // The following is to allow immediate repainting of the list.
5252                   cardPanel.setVisible(false);
5253                   cardMgr.next(cardPanel);
5254
5255                   cardMgr.show(cardPanel, "user");
5256                   cardPanel.setVisible(true);
5257
5258                   // Repaint the table panel if necessary.
5259                   if (tablePanel.isVisible())
5260                   {
5261                      tableCDPanel.setVisible(false);
5262                      tableCDPanel.setVisible(true);
5263                   }
5264
5265                   // If a paid song was just added to the queue and a free song is playing, then kill it.
5266                   if (playerMgr.isCurrentSongFree())
5267                   {
5268                      playerMgr.pressStop();
5269                      setNowPlayingTextField("");
5270                      showCurrentBtn.setEnabled(false);
5271                      treeMgr.setCurrentlyPlayingSong("");
5272                   }
5273                }
5274             }
5275             else
5276             {
5277                logInfo("ERROR: Could not find the requested song! " + strText);
5278
5279
5280                JOptionPane pane   = new JOptionPane("Error:  Could not find the requested song: "+ strText + "...
Please try again.", JOptionPane.ERROR_MESSAGE);
5281                JDialog infoDialog = pane.createDialog(this, "Information");
5282                infoDialog.show();
5283             }
5284
5285             strText = null;
5286             strText = "";
5287             selectionTxtField.setText(strText);
5288
5289             // Force an update.
5290             bButtonsEnabled = false;
5291             checkButtons();
5292
5293             trace("actionPerformed::enterBtn", EXIT);
5294          }
5295          else if (object == adminNextBtn)
5296          {
```

```
5297            trace("actionPerformed::adminNextBtn", ENTER);
5298
5299            playerMgr.pressStop();
5300
5301            setNowPlayingTextField("");
5302
5303            showCurrentBtn.setEnabled(false);
5304            treeMgr.setCurrentlyPlayingSong(" ");
5305
5306            trace("actionPerformed::adminNextBtn", EXIT);
5307         }
5308         else if (object == adminPauseBtn)
5309         {
5310            trace("actionPerformed::adminPauseBtn", ENTER);
5311
5312            playerMgr.pressPause();
5313
5314            trace("actionPerformed::adminPauseBtn", EXIT);
5315         }
5316         else if (object == adminPlayBtn)
5317         {
5318            trace("actionPerformed::adminPlayBtn", ENTER);
5319
5320            int i = playlistList.getMaxSelectionIndex();
5321
5322            if (i >= 0)
5323            {
5324               PlayListEntry mp3 = (PlayListEntry)playlistVector.elementAt(i);
5325
5326               playerMgr.removeFromPlayList(mp3);
5327               playerMgr.addToPlayList(mp3, 0);
5328               playerMgr.pressStop();
5329            }
5330
5331            trace("actionPerformed::adminPlayBtn", EXIT);
5332         }
5333         else if (object == adminMoveUpBtn)
5334         {
5335            trace("actionPerformed::adminMoveUpBtn", ENTER);
5336
5337            int i = playlistList.getMaxSelectionIndex();
5338
5339            if (i > 0)
5340            {
5341               PlayListEntry mp3 = (PlayListEntry)playlistVector.elementAt(i);
5342
5343               playerMgr.removeFromPlayList(mp3);
5344               playerMgr.addToPlayList(mp3, i - 1);
5345
5346               playlistList.setSelectedIndex(i - 1);
5347
5348               playlistVector = playerMgr.getPlayListVector();
5349
5350               playlistList.repaint(playlistList.getVisibleRect());
5351               tree.repaint(tree.getVisibleRect());
5352
5353               playlistList.ensureIndexIsVisible(i - 1);
5354            }
5355
5356            trace("actionPerformed::adminMoveUpBtn", EXIT);
5357         }
5358         else if (object == adminMoveDnBtn)
5359         {
5360            trace("actionPerformed::adminMoveDnBtn", ENTER);
5361
5362            int i = playlistList.getMaxSelectionIndex();
5363
5364            if (i < playlistVector.size() - 1)
5365            {
5366               PlayListEntry mp3 = (PlayListEntry)playlistVector.elementAt(i);
5367
5368               playerMgr.removeFromPlayList(mp3);
5369               playerMgr.addToPlayList(mp3, i + 1);
5370
5371               playlistList.setSelectedIndex(i + 1);
5372
5373               playlistVector = playerMgr.getPlayListVector();
5374
5375               playlistList.repaint(playlistList.getVisibleRect());
5376               tree.repaint(tree.getVisibleRect());
5377
5378               playlistList.ensureIndexIsVisible(i + 1);
5379            }
5380
5381            trace("actionPerformed::adminMoveDnBtn", EXIT);
5382         }
5383         else if (object == adminRemoveBtn)
5384         {
5385            trace("actionPerformed::adminRemoveBtn", ENTER);
5386
```

```
5387            int i = playlistList.getMaxSelectionIndex();
5388
5389            if (i >= 0)
5390            {
5391               PlayListEntry mp3 = (PlayListEntry)playlistVector.elementAt(i);
5392
5393
5394               playerMgr.removeFromPlayList(mp3);
5395
5396
5397               playlistVector = playerMgr.getPlayListVector();
5398
5399               playlistList.repaint(playlistList.getVisibleRect());
5400               tree.repaint(tree.getVisibleRect());
5401            }
5402
5403            trace("actionPerformed::adminRemoveBtn", EXIT);
5404         }
5405         else if (object == ownerIncrementBtn)
5406         {
5407            credits = credits + 1;
5408            newCredits = credits;
5409
5410            intCredits = null;
5411            intCredits = new Integer(credits);
5412
5413            creditsTxtField.setText(intCredits.toString());
5414
5415            checkButtons();
5416
5417            strText = selectionTxtField.getText();
5418
5419            if (credits > 0 && strText.length() == 5)
5420               enterBtn.setEnabled(true);
5421         }
5422         else if (object == ownerDecrementBtn)
5423         {
5424            credits = credits - 1;
5425            newCredits = credits;
5426
5427            intCredits = null;
5428            intCredits = new Integer(credits);
5429
5430            creditsTxtField.setText(intCredits.toString());
5431
5432            checkButtons();
5433
5434            strText = selectionTxtField.getText();
5435
5436            if (credits > 0 && strText.length() == 5)
5437               enterBtn.setEnabled(true);
5438         }
5439         else if (object == ownerDeleteFromDiskBtn)
5440         {
5441            trace("actionPerformed::ownerDeleteFromDiskBtn", ENTER);
5442
5443            int iRow = tree.getMaxSelectionRow();
5444
5445            if (iRow > 0)
5446            {
5447               DefaultMutableTreeNode node;
5448               node = (DefaultMutableTreeNode)tree.getLastSelectedPathComponent();
5449
5450               JFrame confirmationFrame = new JFrame();
5451               ConfirmationDialog confirmationDialog = new ConfirmationDialog(confirmationFrame, "   Delete All
Songs in Node From Disk Confirmation", "Do you wish to delete the following node?: ", node.toString());
5452               int iAnswer = confirmationDialog.getValue();
5453
5454               if (iAnswer == ConfirmationDialog.YES_OPTION)
5455               {
5456                  Vector vector = treeMgr.getAllChildren(node);
5457                  if (vector.size() > 0)
5458                  {
5459                     playerMgr.removeFromPlayList(vector);
5460                     treeMgr.deleteSubTree(node, tree);
5461                     tree.repaint(tree.getVisibleRect());
5462                     playlistList.repaint(playlistList.getVisibleRect());
5463                     playlistVector = playerMgr.getPlayListVector();
5464                     playlistList.setListData(playlistVector);
5465                  }
5466               }
5467            }
5468
5469            trace("actionPerformed::ownerDeleteFromDiskBtn", EXIT);
5470         }
5471         else if (object == ownerAddNodeToQBtn)
5472         {
5473            trace("actionPerformed::ownerAddNodeToQBtn", ENTER);
5474
5475            int iRow = tree.getMaxSelectionRow();
```

```
5477            if (iRow > 0)
5478            {
5479               DefaultMutableTreeNode node;
5480               node = (DefaultMutableTreeNode)tree.getLastSelectedPathComponent();
5481
5482               JFrame confirmationFrame = new JFrame();
5483               ConfirmationDialog confirmationDialog = new ConfirmationDialog(confirmationFrame, "   Add All Songs
in Node to Queue Confirmation", "Do you wish to add the following node to the Queue?: ", node.toString());
5484               int iAnswer = confirmationDialog.getValue();
5485
5486               if (iAnswer == ConfirmationDialog.YES_OPTION)
5487               {
5488                  Vector vector = treeMgr.getAllChildren(node);
5489                  if (vector.size() > 0)
5490                  {
5491                     playerMgr.addToPlayList(vector);
5492
5493                     tree.repaint(tree.getVisibleRect());
5494                     playlistList.repaint(playlistList.getVisibleRect());
5495                     playlistVector = playerMgr.getPlayListVector();
5496                     playlistList.setListData(playlistVector);
5497                  }
5498               }
5499            }
5500
5501            trace("actionPerformed::ownerAddNodeToQBtn", EXIT);
5502         }
5503         else if (object == ownerResetTreeBtn)
5504         {
5505            trace("actionPerformed::ownerResetTreeBtn", ENTER);
5506
5507            int iRow = tree.getMaxSelectionRow();
5508
5509            if (iRow > 0)
5510            {
5511               DefaultMutableTreeNode node;
5512               node = (DefaultMutableTreeNode)tree.getLastSelectedPathComponent();
5513
5514               JFrame confirmationFrame = new JFrame();
5515               ConfirmationDialog confirmationDialog = new ConfirmationDialog(confirmationFrame, "   Reset All Songs
in Node Confirmation", "Do you wish to reset the following node?: ", node.toString());
5516               int iAnswer = confirmationDialog.getValue();
5517
5518               if (iAnswer == ConfirmationDialog.YES_OPTION)
5519               {
5520                  Vector vector = treeMgr.getAllChildren(node);
5521                  if (vector.size() > 0)
5522                  {
5523                     playerMgr.removeFromPlayList(vector);
5524                     treeMgr.resetAll(node);
5525                     tree.repaint(tree.getVisibleRect());
5526                     playlistList.repaint(playlistList.getVisibleRect());
5527                     playlistVector = playerMgr.getPlayListVector();
5528                     playlistList.setListData(playlistVector);
5529                  }
5530               }
5531            }
5532
5533            trace("actionPerformed::ownerResetTreeBtn", EXIT);
5534         }
5535         else if (object == ownerRemNodeFromQBtn)
5536         {
5537            trace("actionPerformed::ownerRemNodeFromQBtn", ENTER);
5538
5539            int iRow = tree.getMaxSelectionRow();
5540
5541            if (iRow > 0)
5542            {
5543               DefaultMutableTreeNode node;
5544               node = (DefaultMutableTreeNode)tree.getLastSelectedPathComponent();
5545
5546               JFrame confirmationFrame = new JFrame();
5547               ConfirmationDialog confirmationDialog = new ConfirmationDialog(confirmationFrame, "   Remove All
Songs in Node from Queue Confirmation", "Do you wish to remove this node from the Queue?: ", node.toString());
5548               int iAnswer = confirmationDialog.getValue();
5549
5550               if (iAnswer == ConfirmationDialog.YES_OPTION)
5551               {
5552                  Vector vector = treeMgr.getAllChildren(node);
5553                  if (vector.size() > 0)
5554                  {
5555                     playerMgr.removeFromPlayList(vector);
5556                     tree.repaint(tree.getVisibleRect());
5557                     playlistList.repaint(playlistList.getVisibleRect());
5558                     playlistVector = playerMgr.getPlayListVector();
5559                     playlistList.setListData(playlistVector);
5560                  }
5561               }
5562            }
5563
```

```
5564            trace("actionPerformed::ownerRemNodeFromQBtn", EXIT);
5565         }
5566         else if (object == adminShowQueuedCB)
5567         {
5568            trace("actionPerformed::adminShowQueuedCB", ENTER);
5569
5570            if (adminShowQueuedCB.isSelected())
5571            {
5572               bShowQueued = true;
5573            }
5574            else
5575            {
5576               bShowQueued = false;
5577            }
5578
5579            trace("actionPerformed::adminShowQueuedCB", EXIT);
5580         }
5581         else if (object == adminShowConfirmationCB)
5582         {
5583            trace("actionPerformed::adminShowConfirmationCB", ENTER);
5584
5585            if (adminShowConfirmationCB.isSelected())
5586            {
5587               bShowConfirmation = true;
5588            }
5589            else
5590            {
5591               bShowConfirmation = false;
5592            }
5593
5594            trace("actionPerformed::adminShowConfirmationCB", EXIT);
5595         }
5596         else if (object == adminRandomPlayCB)
5597         {
5598            trace("actionPerformed::adminRandomPlayCB", ENTER);
5599
5600            if (adminRandomPlayCB.isSelected())
5601            {
5602               bRandomPlay = true;
5603               adminRandomIntervalLabel.setVisible(true);
5604               adminRandomIntervalSB.setVisible(true);
5605            }
5606            else
5607            {
5608               bRandomPlay = false;
5609               adminRandomIntervalLabel.setVisible(false);
5610               adminRandomIntervalSB.setVisible(false);
5611            }
5612
5613            trace("actionPerformed::adminRandomPlayCB", EXIT);
5614         }
5615         else if (object == ownerAddPathBtn)
5616         {
5617            menuAddDir_doWork();
5618
5619
5620            // Artificially bump up the user inactivity time, so that at 600 ticks (300 seconds),
5621            // the "Popular" tables will be regenerated using the new CDs that were just added
5622            // to the system.
5623            iElapsedUserInactivity = 596;
5624            bDirtyFlag = true;
5625
5626
5627            initCDVector();
5628            initGenreVector();
5629            initGenreTitleVector();
5630
5631
5632            // Update the Genre Panel.
5633            genreTitleVect = new Vector();
5634            for (Enumeration enum = GenreTitleVector.elements(); enum.hasMoreElements(); )
5635            {
5636               Vector rowVect = (Vector)enum.nextElement();
5637               genreTitleVect.addElement((String)rowVect.elementAt(0));
5638            }
5639            genreList.setListData(genreTitleVect);
5640
5641
5642            if (bFlipToRandom == true)
5643            {
5644               flipToRandomCD();
5645            }
5646
5647            initClassicPanel(CURRENT);
5648            addToClassicPanel();
5649            updateVisibleCDTextField();
5650            checkScrollButtons();
5651         }
5652         else if (object == tableSongViewBtn)
5653         {
```

```
5654            trace("actionPerformed::tableSongViewBtn", ENTER);
5655
5656            bTableSongView = true;
5657            tableSongViewBtn.setEnabled(false);
5658            tableCDViewBtn.setEnabled(true);
5659
5660
5661            iTableSize = TABLE_TOP50;
5662            tableShowTop50Btn.setEnabled(false);
5663            tableShowTop100Btn.setEnabled(true);
5664            tableShowAllBtn.setEnabled(true);
5665            tableShowNewBtn.setEnabled(true);
5666
5667
5668            tableShowNewBtn.setVisible(false);
5669
5670            tablePanel.setVisible(false);
5671
5672            if (tableCDPanel != null)
5673            {
5674               tablePanel.remove(tableCDPanel);
5675               tableCDPanel = null;
5676            }
5677            tablePanel.setVisible(true);
5678
5679            showTable();
5680
5681            trace("actionPerformed::tableSongViewBtn", EXIT);
5682         }
5683         else if (object == tableCDViewBtn)
5684         {
5685            trace("actionPerformed::tableCDViewBtn", ENTER);
5686
5687            bTableSongView = false;
5688            tableCDViewBtn.setEnabled(false);
5689            tableSongViewBtn.setEnabled(true);
5690
5691            tableShowNewBtn.setVisible(true);
5692
5693            tablePanel.setVisible(false);
5694            if (tableCDPanel != null)
5695            {
5696               tablePanel.remove(tableCDPanel);
5697               tableCDPanel = null;
5698            }
5699            tablePanel.setVisible(true);
5700
5701            showTable();
5702
5703            trace("actionPerformed::tableCDViewBtn", EXIT);
5704         }
5705         else if (object == tableAbsRankingBtn)
5706         {
5707            trace("actionPerformed::tableAbsRankingBtn", ENTER);
5708
5709            bTableAbsRanking = true;
5710            tableAbsRankingBtn.setEnabled(false);
5711            tablePwrRankingBtn.setEnabled(true);
5712
5713            showTable();
5714
5715            trace("actionPerformed::tableAbsRankingBtn", EXIT);
5716         }
5717         else if (object == tablePwrRankingBtn)
5718         {
5719            trace("actionPerformed::tablePwrRankingBtn", ENTER);
5720
5721            bTableAbsRanking = false;
5722            tablePwrRankingBtn.setEnabled(false);
5723            tableAbsRankingBtn.setEnabled(true);
5724
5725            showTable();
5726
5727            trace("actionPerformed::tablePwrRankingBtn", EXIT);
5728         }
5729         else if (object == tableShowTop50Btn)
5730         {
5731            trace("actionPerformed::tableShowTop50Btn", ENTER);
5732
5733            iTableSize = TABLE_TOP50;
5734
5735            tableShowTop50Btn.setEnabled(false);
5736            tableShowTop100Btn.setEnabled(true);
5737            tableShowAllBtn.setEnabled(true);
5738            tableShowNewBtn.setEnabled(true);
5739
5740            showTable();
5741
5742            trace("actionPerformed::tableShowTop50Btn", EXIT);
5743         }
```

```
5744         else if (object == tableShowTop100Btn)
5745         {
5746            trace("actionPerformed::tableShowTop100Btn", ENTER);
5747
5748            iTableSize = TABLE_TOP100;
5749
5750            tableShowTop50Btn.setEnabled(true);
5751            tableShowTop100Btn.setEnabled(false);
5752            tableShowAllBtn.setEnabled(true);
5753            tableShowNewBtn.setEnabled(true);
5754
5755            showTable();
5756
5757            trace("actionPerformed::tableShowTop100Btn", EXIT);
5758         }
5759         else if (object == tableShowAllBtn)
5760         {
5761            trace("actionPerformed::tableShowAllBtn", ENTER);
5762
5763            iTableSize = TABLE_ALL;
5764
5765            tableShowTop50Btn.setEnabled(true);
5766            tableShowTop100Btn.setEnabled(true);
5767            tableShowAllBtn.setEnabled(false);
5768            tableShowNewBtn.setEnabled(true);
5769
5770            showTable();
5771
5772            trace("actionPerformed::tableShowAllBtn", EXIT);
5773         }
5774         else if (object == tableShowNewBtn)
5775         {
5776            trace("actionPerformed::tableShowNewBtn", ENTER);
5777
5778            iTableSize = TABLE_NEW;
5779
5780            tableShowTop50Btn.setEnabled(true);
5781            tableShowTop100Btn.setEnabled(true);
5782            tableShowAllBtn.setEnabled(true);
5783            tableShowNewBtn.setEnabled(false);
5784
5785            showTable();
5786
5787            trace("actionPerformed::tableShowNewBtn", EXIT);
5788         }
5789
5790         //trace("actionPerformed()", EXIT);
5791      }
5792
5793      private void loadLogFile()
5794      {
5795         trace("loadLogFile()", ENTER);
5796
5797         // Read in contents of the .log file and put it into the log text area.
5798         try
5799         {
5800            StringBuffer strBuffer = new StringBuffer();
5801            String strTemp = new String();
5802
5803            if (logFile.exists())
5804            {
5805               BufferedReader in = new BufferedReader(new FileReader(logFile));
5806
5807               // Do a priming read.
5808               strTemp = in.readLine();
5809
5810               while (strTemp != null)
5811               {
5812                  strBuffer.append(strTemp);
5813                  strBuffer.append('\r');
5814                  strBuffer.append('\n');
5815
5816                  strTemp = in.readLine();
5817               }
5818
5819               in.close();
5820
5821               adminLogTextArea.setText(strBuffer.toString());
5822            }
5823         }
5824         catch (java.io.IOException e)
5825         {
5826            System.out.println("Could not read MP3Jukeboxx.log!");
5827         }
5828
5829         trace("loadLogFile()", EXIT);
5830      }
5831
5832      public void checkScrollButtons()
5833      {
```

```
5834        trace("checkScrollButtons", ENTER);
5835
5836
5837        if (iCurrentCDPtr + 4 <= iMaxCDPtr)
5838        {
5839           btmBtn.setEnabled(true);
5840           nextPageBtn.setEnabled(true);
5841        }
5842        else
5843        {
5844           btmBtn.setEnabled(false);
5845           nextPageBtn.setEnabled(false);
5846        }
5847
5848
5849        if (iCurrentCDPtr > 0)
5850        {
5851           topBtn.setEnabled(true);
5852           prevPageBtn.setEnabled(true);
5853        }
5854        else
5855        {
5856           topBtn.setEnabled(false);
5857           prevPageBtn.setEnabled(false);
5858        }
5859
5860
5861        if (iCurrentGenrePtr > 0)
5862           prevGenreBtn.setEnabled(true);
5863        else
5864           prevGenreBtn.setEnabled(false);
5865
5866
5867        if (iCurrentGenrePtr < iMaxGenrePtr)
5868           nextGenreBtn.setEnabled(true);
5869        else
5870           nextGenreBtn.setEnabled(false);
5871
5872        checkButtons();
5873
5874        trace("checkScrollButtons", EXIT);
5875     }
5876
5877     public void checkScrollButtons(int iCDPtr)
5878     {
5879        trace("checkScrollButtons(int)", ENTER);
5880
5881        if (iCDPtr < iMaxCDPtr)
5882        {
5883           btmBtn.setEnabled(true);
5884           nextPageBtn.setEnabled(true);
5885        }
5886        else
5887        {
5888           btmBtn.setEnabled(false);
5889           nextPageBtn.setEnabled(false);
5890        }
5891
5892
5893        if (iCDPtr > 0)
5894        {
5895           topBtn.setEnabled(true);
5896           prevPageBtn.setEnabled(true);
5897        }
5898        else
5899        {
5900           topBtn.setEnabled(false);
5901           prevPageBtn.setEnabled(false);
5902        }
5903
5904
5905        int iGenrePtr = getGenrePtrForSelectedCD();
5906
5907        if (iGenrePtr > 0)
5908           prevGenreBtn.setEnabled(true);
5909        else
5910           prevGenreBtn.setEnabled(false);
5911
5912
5913        if (iGenrePtr < iMaxGenrePtr)
5914           nextGenreBtn.setEnabled(true);
5915        else
5916           nextGenreBtn.setEnabled(false);
5917
5918        checkButtons();
5919
5920        trace("checkScrollButtons(int)", EXIT);
5921     }
5922
5923     public void disableBottomPanel()
```

```
5924    {
5925       trace("disableBottomPanel()", ENTER);
5926
5927       topBtn.setEnabled(false);
5928       nextGenreBtn.setEnabled(false);
5929       prevGenreBtn.setEnabled(false);
5930       btmBtn.setEnabled(false);
5931       prevPageBtn.setEnabled(false);
5932       nextPageBtn.setEnabled(false);
5933
5934       showCurrentBtn.setEnabled(false);
5935
5936       tableBtn.setEnabled(false);
5937       genreBtn.setEnabled(false);
5938       srchBtn.setEnabled(false);
5939
5940       btn_0.setEnabled(false);
5941       btn_1.setEnabled(false);
5942       btn_2.setEnabled(false);
5943       btn_3.setEnabled(false);
5944       btn_4.setEnabled(false);
5945       btn_5.setEnabled(false);
5946       btn_6.setEnabled(false);
5947       btn_7.setEnabled(false);
5948       btn_8.setEnabled(false);
5949       btn_9.setEnabled(false);
5950       enterBtn.setEnabled(false);
5951       cancelBtn.setEnabled(false);
5952
5953       trace("disableBottomPanel()", EXIT);
5954    }
5955
5956    public void checkBottomPanel()
5957    {
5958       trace("checkBottomPanel()", ENTER);
5959
5960       topBtn.setEnabled(true);
5961       nextGenreBtn.setEnabled(true);
5962       prevGenreBtn.setEnabled(true);
5963       btmBtn.setEnabled(true);
5964       prevPageBtn.setEnabled(true);
5965       nextPageBtn.setEnabled(true);
5966
5967       if (playerMgr.getStatus() == PlayerMgr.STOPPED)
5968          showCurrentBtn.setEnabled(false);
5969       else
5970          showCurrentBtn.setEnabled(true);
5971
5972       switch (iVisiblePanel)
5973       {
5974          case TABLE_PANEL:
5975             tableBtn.setEnabled(false);
5976             genreBtn.setEnabled(true);
5977             srchBtn.setEnabled(true);
5978
5979             topBtn.setEnabled(false);
5980             nextGenreBtn.setEnabled(false);
5981             prevGenreBtn.setEnabled(false);
5982             btmBtn.setEnabled(false);
5983             prevPageBtn.setEnabled(false);
5984             nextPageBtn.setEnabled(false);
5985             showCurrentBtn.setEnabled(false);
5986             break;
5987
5988          case SEARCH_PANEL:
5989             srchBtn.setEnabled(false);
5990             genreBtn.setEnabled(true);
5991             tableBtn.setEnabled(true);
5992             break;
5993
5994          case GENRE_PANEL:
5995             genreBtn.setEnabled(false);
5996             srchBtn.setEnabled(true);
5997             tableBtn.setEnabled(true);
5998
5999             topBtn.setEnabled(false);
6000             nextGenreBtn.setEnabled(false);
6001             prevGenreBtn.setEnabled(false);
6002             btmBtn.setEnabled(false);
6003             prevPageBtn.setEnabled(false);
6004             nextPageBtn.setEnabled(false);
6005             showCurrentBtn.setEnabled(false);
6006             break;
6007
6008          default:
6009             genreBtn.setEnabled(true);
6010             srchBtn.setEnabled(true);
6011             tableBtn.setEnabled(true);
6012             break;
6013       }
```

```
6015          updateVisibleCDTextField();
6016          checkScrollButtons();
6017
6018          trace("checkBottomPanel()", EXIT);
6019       }
6020
6021       public void checkButtons()
6022       {
6023          trace("checkButtons()", ENTER);
6024
6025          strText = selectionTxtField.getText();
6026
6027          if (strText.length() > 0)
6028          {
6029             cancelBtn.setEnabled(true);
6030          }
6031          else
6032          {
6033             cancelBtn.setEnabled(false);
6034          }
6035
6036          if (credits > 0)
6037          {
6038             if (strText.length() == 5)
6039                toggleButtons(false);
6040             else
6041                toggleButtons(true);
6042          }
6043          else
6044          {
6045             // Force an update.
6046             bButtonsEnabled = true;
6047             toggleButtons(false);
6048          }
6049
6050          bEntryButtonsStale = true;
6051
6052          trace("checkButtons()", EXIT);
6053       }
6054
6055       public void checkSearchScrollButtons()
6056       {
6057          trace("checkSearchScrollButtons()", ENTER);
6058
6059          if (searchVector != null && searchVector.size() > 0)
6060          {
6061             int i = searchTable.getSelectionModel().getLeadSelectionIndex();
6062
6063             if (i == 0)
6064                searchPageUpBtn.setEnabled(false);
6065             else
6066                searchPageUpBtn.setEnabled(true);
6067
6068
6069             if (i == searchVector.size() - 1)
6070                searchPageDnBtn.setEnabled(false);
6071             else
6072                searchPageDnBtn.setEnabled(true);
6073          }
6074          else
6075          {
6076             searchPageUpBtn.setEnabled(false);
6077             searchPageDnBtn.setEnabled(false);
6078          }
6079
6080          trace("checkSearchScrollButtons()", EXIT);
6081       }
6082
6083       public void checkTableScrollButtons()
6084       {
6085          trace("checkTableScrollButtons()", ENTER);
6086
6087          if (tableVector != null && tableVector.size() > 0)
6088          {
6089             int i = table.getSelectionModel().getLeadSelectionIndex();
6090
6091             if (i == 0)
6092                tablePageUpBtn.setEnabled(false);
6093             else
6094                tablePageUpBtn.setEnabled(true);
6095
6096
6097             if (i == tableVector.size() - 1)
6098                tablePageDnBtn.setEnabled(false);
6099             else
6100                tablePageDnBtn.setEnabled(true);
6101          }
6102          else
6103          {
6104             tablePageUpBtn.setEnabled(false);
```

```
6105            tablePageDnBtn.setEnabled(false);
6106         }
6107
6108         trace("checkTableScrollButtons()", EXIT);
6109      }
6110
6111      public void timer_actionPerformed()
6112      {
6113         //trace("timer_actionPerformed()", ENTER);
6114
6115         iElapsedRuntime = iElapsedRuntime + 1;
6116
6117
6118
6119         // TDM: TEMPORARY!!!! Need to figure out how to fire an event from the SpinButtons.
6120         if (iVisiblePanel == ADMIN_PANEL)
6121         {
6122            int iNum = ownerNumToQueueSB.getValue();
6123            if (iNum != iNumberToQueue)
6124            {
6125               iNumberToQueue = iNum;
6126               intNumberToQueue = new Integer(iNumberToQueue);
6127               playerMgr.setNumberToQueue(iNumberToQueue);
6128               playlistList.setListData(playlistVector);
6129               playlistVector = playerMgr.getPlayListVector();
6130               playlistList.repaint(playlistList.getVisibleRect());
6131            }
6132
6133            int iVol = adminPlayerVolumeSB.getValue();
6134            if (iVol != iPlayerVolume)
6135            {
6136               playerMgr.setVolume(adminPlayerVolumeSB.getValue());
6137            }
6138
6139            int iVal = adminRandomIntervalSB.getValue();
6140            if (iVal != iRandomPlayInterval)
6141            {
6142               iRandomPlayInterval = iVal;
6143               intRandomPlayInterval = new Integer(iRandomPlayInterval);
6144            }
6145         }
6146
6147         // Get the current state of the acceptor.  Display any messages if necc.
6148         String strState = gbaMgr.getCurrentState();
6149         if (strState.equalsIgnoreCase("JAMMED"))
6150         {
6151            logInfo("GBAMGR:  Jammed bill encountered.");
6152
6153            JOptionPane pane   = new JOptionPane("A jammed bill has been encountered.  Please notify mgmt.",
JOptionPane.ERROR_MESSAGE);
6154            JDialog infoDialog = pane.createDialog(this, "Error:  Jammed Bill");
6155            infoDialog.show();
6156         }
6157         else if (strState.equalsIgnoreCase("FULL"))
6158         {
6159            logInfo("GBAMGR:  Stacker Full condition encountered.");
6160
6161            JOptionPane pane   = new JOptionPane("The stacker is full.  Please notify mgmt.", JOptionPane.
ERROR_MESSAGE);
6162            JDialog infoDialog = pane.createDialog(this, "Error:  Stacker Full");
6163            infoDialog.show();
6164         }
6165         else if (strState.equalsIgnoreCase("FAILURE"))
6166         {
6167            logInfo("GBAMGR:  A general failure with the bill validator has occured.");
6168
6169            JOptionPane pane   = new JOptionPane("A general failure has occurred.  Please notify mgmt.", JOptionPane
.ERROR_MESSAGE);
6170            JDialog infoDialog = pane.createDialog(this, "Error:  Acceptor Failure");
6171            infoDialog.show();
6172         }
6173
6174
6175         // Get the last event detected by the acceptor.  First, see if there was a cheat attempt.
6176         String strLastEvent = gbaMgr.getLastEvent();
6177
6178         if (strLastEvent.equalsIgnoreCase("CHEATED"))
6179         {
6180            logInfo("GBAMGR:  An attempt to cheat the bill validator has been detected.");
6181
6182            JOptionPane pane   = new JOptionPane("An attempt to cheat the bill validator has been detected.  Please
notify mgmt.", JOptionPane.ERROR_MESSAGE);
6183            JDialog infoDialog = pane.createDialog(this, "Error:  Cheater");
6184            infoDialog.show();
6185         }
6186         else if (strLastEvent.equalsIgnoreCase("REJECTED"))
6187         {
6188            logInfo("GBAMGR:  An invalid bill has been detected.");
6189         }
6190         else if (strLastEvent.equals("STACKED"))
```

```
6191        {
6192           // Next, see if the user added any credits.  If so, enable the selection buttons.
6193           String strLastBill  = gbaMgr.getLastBillProcessed();
6194
6195           if (strLastBill.equals("UNKNOWN") == false)
6196           {
6197              if (strLastBill.equals("ONE"))
6198              {
6199                 newCredits = credits + iCreditsPer;
6200                 logStackedBill(1);
6201              }
6202              else if (strLastBill.equals("TWO"))
6203              {
6204                 newCredits = credits + (2 * iCreditsPer);
6205                 logStackedBill(2);
6206              }
6207              else if (strLastBill.equals("FIVE"))
6208              {
6209                 newCredits = credits + (5 * iCreditsPer);
6210                 logStackedBill(5);
6211              }
6212              else if (strLastBill.equals("TEN"))
6213              {
6214                 newCredits = credits + (10 * iCreditsPer);
6215                 logStackedBill(10);
6216              }
6217              else if (strLastBill.equals("TWENTY"))
6218              {
6219                 newCredits = credits + (20 * iCreditsPer);
6220                 logStackedBill(20);
6221              }
6222              else if (strLastBill.equals("FIFTY"))
6223              {
6224                 newCredits = credits + (50 * iCreditsPer);
6225                 logStackedBill(50);
6226              }
6227              else if (strLastBill.equals("HUNDRED"))
6228              {
6229                 newCredits = credits + (100 * iCreditsPer);
6230                 logStackedBill(100);
6231              }
6232           }
6233        }
6234
6235
6236        // If the user has entered more than one 1$ bill or entered any 2$, 5$, 10$, $20, or $100
6237        // bill, then give them bonus credits if they are configured.
6238        if (credits != newCredits)
6239        {
6240           int iCurrBonusFactor = 0;
6241
6242           if (newCredits >= iBonusLevel_4)
6243              iCurrBonusFactor = iBonusFactor_4;
6244           else if (newCredits >= iBonusLevel_3)
6245              iCurrBonusFactor = iBonusFactor_3;
6246           else if (newCredits >= iBonusLevel_2)
6247              iCurrBonusFactor = iBonusFactor_2;
6248           else if (newCredits >= iBonusLevel_1)
6249              iCurrBonusFactor = iBonusFactor_1;
6250
6251           newCredits = newCredits + iCurrBonusFactor;
6252
6253
6254           logInfo("Credits= " + newCredits + "  Old credits=" + credits);
6255
6256           credits = newCredits;
6257
6258           intCredits = null;
6259           intCredits = new Integer(credits);
6260
6261           creditsTxtField.setText(intCredits.toString());
6262
6263           checkButtons();
6264
6265           strText = selectionTxtField.getText();
6266
6267           if (credits > 0 && strText.length() == 5)
6268              enterBtn.setEnabled(true);
6269        }
6270
6271
6272        // Enable/Disable the selection entry buttons if necessary.
6273        strText = selectionTxtField.getText();
6274        if (strText.length() > 0 && bEntryButtonsStale == true)
6275        {
6276           bEntryButtonsStale = false;
6277
6278           if (strText.length() == 5)
6279              toggleButtons(false);
6280           else
```

```
6281            toggleButtons(true);
6282        }
6283
6284
6285
6286        // TDM: Need to periodically serialize the number of credits in the system!!!
6287
6288
6289
6290        // Save the properties (which contains the number of credits) to disk. (every 5 minutes)
6291        if ((iElapsedRuntime % 600) == 0 && credits > 0)
6292        {
6293           saveProperties();
6294        }
6295
6296
6297
6298        // Perform Garbage Collection every 5 minutes.
6299        if (iElapsedRuntime % 600 == 0)
6300        {
6301           trace(" ");
6302           trace("+timer_actionPerformed:: (5 minute interval)");
6303           trace("Free memory before gc(): " + new Long(Runtime.getRuntime().freeMemory()).toString());
6304           System.gc();
6305           trace("Free memory after gc():  " + new Long(Runtime.getRuntime().freeMemory()).toString());
6306           trace("-timer_actionPerformed:: (5 minute interval)");
6307           trace(" ");
6308        }
6309
6310
6311        // Serialize the tree data every 5 minutes so that the played/queued information
6312        // will always be up to date (at most, off by a song or two).
6313        if ((iElapsedRuntime % 600) == 0 && bDirtyFlag == true && bIsAppFunctional == true)
6314        {
6315           iElapsedRuntime = 1;
6316
6317           treeMgr.writeTreeToDisk();
6318
6319           // If there are any songs in the queue, then save it to disk.
6320           if (playerMgr.getQueuedSongCount() > 0)
6321           {
6322              savePlaylist();
6323           }
6324           else
6325           {
6326              File file = new File("MP3Jukeboxx.PL");
6327
6328              if (!file.exists())
6329              {
6330                 file = null;
6331                 file = new File("c:/kiosk/MP3Jukeboxx.PL");
6332              }
6333
6334              if (file.exists())
6335              {
6336                 try
6337                 {
6338                    file.delete();
6339                 }
6340                 catch (java.lang.SecurityException e)
6341                 {
6342                    logException(e);
6343                    e.printStackTrace();
6344                 }
6345              }
6346           }
6347        }
6348
6349
6350        // Update the "Most Popular" table if there's been 5 minutes of user inactivity and
6351        // if a song has been played since the last time we dumped the stats and refreshed the "popular" table.
6352        if ((iElapsedUserInactivity % 600) == 0 && bDirtyFlag == true)
6353        {
6354           trace("+timer_actionPerformed:: 5 min. User Inactivity");
6355
6356           iElapsedUserInactivity = 1;
6357           bDirtyFlag = false;
6358
6359           System.out.println("Re-initializing popular table vectors.");
6360           initTableVectors();
6361
6362           System.out.println("Dumping CD Stats to disk");
6363           treeMgr.dumpCDStats();
6364
6365           /*
6366           tableVector = top50Vect;
6367
6368           iTableSize = TABLE_TOP50;
6369           tableShowTop50Btn.setEnabled(false);
6370           tableShowTop100Btn.setEnabled(true);
```

```
6371            tableShowAllBtn.setEnabled(true);
6372            tableShowNewBtn.setEnabled(true);
6373
6374            bTableSongView = true;
6375            tableSongViewBtn.setEnabled(false);
6376            tableCDViewBtn.setEnabled(true);
6377            tableShowNewBtn.setVisible(false);
6378
6379            showTable();
6380            */
6381
6382            trace("-timer_actionPerformed:: 5 min. User Inactivity");
6383         }
6384
6385
6386
6387         int iStatus = playerMgr.getStatus();
6388         if (iStatus == PlayerMgr.STOPPED)
6389         {
6390            if (!nowPlayingTxtField.getText().equalsIgnoreCase(""))
6391            {
6392               setNowPlayingTextField("");
6393
6394               showCurrentBtn.setEnabled(false);
6395               treeMgr.setCurrentlyPlayingSong(" ");
6396            }
6397
6398            iElapsedSilence = iElapsedSilence + 1;
6399
6400            playlistVector = playerMgr.getPlayListVector();
6401            playlistList.setListData(playlistVector);
6402
6403            playlistList.repaint(playlistList.getVisibleRect());
6404            tree.repaint(tree.getVisibleRect());
6405         }
6406         else
6407         {
6408            // Update the currently playing song text field.
6409            if (nowPlayingTxtField.getText().equalsIgnoreCase(""))
6410            {
6411               treeMgr.setCurrentlyPlayingSong(playerMgr.getCurrentSong());
6412
6413               if (iVisiblePanel == CLASSIC_PANEL)
6414                  showCurrentBtn.setEnabled(true);
6415
6416
6417               playlistVector = playerMgr.getPlayListVector();
6418               playlistList.setListData(playlistVector);
6419
6420               playlistList.repaint(playlistList.getVisibleRect());
6421               tree.repaint(tree.getVisibleRect());
6422
6423               repaintCDPanels();
6424
6425
6426               String strSelNum = null;
6427               PlayListEntry mp3 = playerMgr.getCurrentPlayListObject();
6428               int iTrackNum = mp3.getTrackNum();
6429               if (iTrackNum < 10)
6430               {
6431                  strSelNum = treeMgr.getCDNumberForSong(tree, mp3) + "0" + Integer.toString(iTrackNum);
6432               }
6433               else
6434               {
6435                  strSelNum = treeMgr.getCDNumberForSong(tree, mp3) + Integer.toString(iTrackNum);
6436               }
6437               String strNowPlaying = strSelNum + " - " + mp3.getArtist() + mp3.getSong() + "  (" + mp3.getGenre() +
")";
6438
6439
6440               if (playerMgr.isCurrentSongFree())
6441               {
6442                  setNowPlayingTextField(strNowPlaying + " (free)");
6443               }
6444               else
6445               {
6446                  setNowPlayingTextField(strNowPlaying);
6447               }
6448
6449               logInfo("Playing: " + nowPlayingTxtField.getText());
6450            }
6451
6452            iElapsedSilence = 1;
6453         }
6454
6455
6456         // Play a random free song if there has been a period of silence.
6457         if (bRandomPlay == true)
6458         {
6459            if ((iElapsedSilence % ((iRandomPlayInterval * 60) * 2)) == 0)
```

```
6460            {
6461               if (bDebug)
6462               {
6463                  System.out.println(iRandomPlayInterval + "minutes of inactivity");
6464               }
6465
6466               if (playerMgr.getQueuedSongCount() == 0)
6467               {
6468                  PlayListEntry mp3 = treeMgr.next(tree);
6469
6470                  playerMgr.addToPlayList(mp3);
6471               }
6472            }
6473         }
6474
6475
6476         // If configured to have a non-empty queue, then play/add a free song if the queue is empty/below the
threshold.
6477         if (playerMgr.isReadyForNextMp3())
6478         {
6479            PlayListEntry mp3 = treeMgr.next(tree);
6480
6481            playerMgr.addToPlayList(mp3);
6482         }
6483
6484         //trace("timer_actionPerformed()", EXIT);
6485      }
6486
6487      private void setNowPlayingTextField(String strText)
6488      {
6489         trace("setNowPlayingTextField()", ENTER);
6490
6491         // If a long title, ensure that the left-most portion of the string (containing the selection)
6492         // is visible.
6493
6494         nowPlayingTxtField.setText(strText);
6495         nowPlayingTxtField.setScrollOffset(0);
6496         nowPlayingTxtField.setCaretPosition(0);
6497
6498         trace("setNowPlayingTextField()", EXIT);
6499      }
6500
6501      private void enableFunctionality()
6502      {
6503         trace("enableFunctionality()", ENTER);
6504
6505         bIsAppStarted = true;
6506
6507         // Start playerMgr, which will do its thing in a separate thread.
6508         playerMgr.start();
6509
6510         // Start the timer.
6511         timer.start();
6512
6513         playerMgr.setLockOnQueue(false);
6514         playerMgr.releaseInitialLock();
6515
6516         // Enable everything.
6517         tree.setEnabled(true);
6518
6519         this.setEnabled(true);
6520
6521         trace("enableFunctionality()", EXIT);
6522      }
6523
6524      private void repaintCDPanels()
6525      {
6526         trace("repaintCDPanels()", ENTER);
6527
6528         if (northwestCD != null)
6529            northwestCD.forceRepaint();
6530
6531         if (northeastCD != null)
6532            northeastCD.forceRepaint();
6533
6534         if (southwestCD != null)
6535            southwestCD.forceRepaint();
6536
6537         if (southeastCD != null)
6538            southeastCD.forceRepaint();
6539
6540         if (genreNorthCD != null)
6541            genreNorthCD.forceRepaint();
6542
6543         if (genreSouthCD != null)
6544            genreSouthCD.forceRepaint();
6545
6546         if (tableCDPanel != null)
6547            tableCDPanel.forceRepaint();
6548
```

```
6549          if (searchCDPanel != null)
6550             searchCDPanel.forceRepaint();
6551
6552          trace("repaintCDPanels()", EXIT);
6553       }
6554
6555       private void toggleButtons(boolean bEnable)
6556       {
6557          trace("toggleButtons()", ENTER);
6558
6559          if (bEnable != bButtonsEnabled)
6560          {
6561             if (bEnable == true)
6562             {
6563                btn_0.setEnabled(true);
6564                btn_1.setEnabled(true);
6565                btn_2.setEnabled(true);
6566                btn_3.setEnabled(true);
6567                btn_4.setEnabled(true);
6568                btn_5.setEnabled(true);
6569                btn_6.setEnabled(true);
6570                btn_7.setEnabled(true);
6571                btn_8.setEnabled(true);
6572                btn_9.setEnabled(true);
6573
6574                enterBtn.setEnabled(false);
6575
6576                bButtonsEnabled = true;
6577             }
6578             else
6579             {
6580                btn_0.setEnabled(false);
6581                btn_1.setEnabled(false);
6582                btn_2.setEnabled(false);
6583                btn_3.setEnabled(false);
6584                btn_4.setEnabled(false);
6585                btn_5.setEnabled(false);
6586                btn_6.setEnabled(false);
6587                btn_7.setEnabled(false);
6588                btn_8.setEnabled(false);
6589                btn_9.setEnabled(false);
6590
6591                strText = selectionTxtField.getText();
6592
6593                if (credits > 0 && strText.length() == 5)
6594                   enterBtn.setEnabled(true);
6595                else
6596                   enterBtn.setEnabled(false);
6597
6598                bButtonsEnabled = false;
6599             }
6600          }
6601
6602          trace("toggleButtons()", EXIT);
6603       }
6604
6605       public void logAppExceptionInfo()
6606       {
6607          trace("logAppExceptionInfo()", ENTER);
6608
6609          logInfo("Application Exception Info:");
6610          logInfo("---------------------------");
6611          logInfo("iMaxCDPtr:         " + iMaxCDPtr);
6612          logInfo("iMaxGenrePtr:      " + iMaxGenrePtr);
6613          logInfo("iCurrentCDPtr:     " + iCurrentCDPtr);
6614          logInfo("iCurrentGenrePtr:  " + iCurrentGenrePtr);
6615          logInfo("Visible CDs:       " + visibleCDsTxtField.getText());
6616
6617          trace("logAppExceptionInfo()", EXIT);
6618       }
6619
6620       public void cleanup()
6621       {
6622          trace("cleanup()", ENTER);
6623
6624          playerMgr.setLockOnQueue(true);
6625
6626          // Save the current playlist regardless of whether there are any songs in the queue.
6627          savePlaylist();
6628
6629          // Stop the timer in this thread.
6630          timer.stop();
6631
6632          // Serialize the tree to disk.
6633          treeMgr.writeTreeToDisk();
6634
6635          // Save application settings to disk.
6636          saveProperties();
6637
6638          // Save bill tracking data to disk (as a vector object to thwart foul-play).
```

```
6639         saveAcceptorVector();
6640
6641         trace("cleanup()", EXIT);
6642      }
6643
6644      public void savePlaylist()
6645      {
6646         trace("savePlaylist()", ENTER);
6647
6648         // Next, serialize the current playlist to disk.
6649         File file = new File("MP3Jukeboxx.PL");
6650         try
6651         {
6652            playerMgr.savePlayList(file);
6653         }
6654         catch (java.io.FileNotFoundException eNotFound)
6655         {
6656            file = new File("c:/kiosk/MP3Jukeboxx.PL");
6657
6658            try
6659            {
6660               playerMgr.savePlayList(file);
6661            }
6662            catch (java.io.FileNotFoundException eNotFound2)
6663            {
6664               logException(eNotFound2);
6665               System.out.println("Could not save playlist to: " + file.toString());
6666            }
6667         }
6668         trace("savePlaylist()", EXIT);
6669      }
6670
6671      public void windowClosing(WindowEvent e)
6672      {
6673         logInfo("Exiting...");
6674
6675         System.exit(0);
6676      }
6677
6678      public void windowIconified(java.awt.event.WindowEvent e) { }
6679      public void windowDeiconified(java.awt.event.WindowEvent e) { }
6680      public void windowDeactivated(java.awt.event.WindowEvent e) { }
6681      public void windowOpened(java.awt.event.WindowEvent e) { }
6682      public void windowClosed(java.awt.event.WindowEvent e) { }
6683      public void windowActivated(java.awt.event.WindowEvent e) { }
6684
6685      public void mousePressed(java.awt.event.MouseEvent e) { }
6686      public void mouseReleased(java.awt.event.MouseEvent e) { }
6687      public void mouseEntered(java.awt.event.MouseEvent e) { }
6688      public void mouseExited(java.awt.event.MouseEvent e) { }
6689
6690      public void mouseClicked(java.awt.event.MouseEvent e)
6691      {
6692         if (e.getClickCount() == 2)
6693         {
6694            Object object = e.getSource();
6695
6696            if (object == totalCDsTxtField)
6697            {
6698               setColumnVisibility();
6699            }
6700         }
6701      }
6702
6703      private void setColumnVisibility()
6704      {
6705         trace("setColumnVisibility()", ENTER);
6706
6707         if (bPopularTableColumnsHidden == false)
6708         {
6709            bPopularTableColumnsHidden = true;
6710
6711            table.getColumnModel().removeColumn(tcAge);
6712            table.getColumnModel().removeColumn(tcPlays);
6713            table.getColumnModel().removeColumn(tcPlaysPerDay);
6714         }
6715         else
6716         {
6717            bPopularTableColumnsHidden = false;
6718
6719            table.getColumnModel().addColumn(tcPlaysPerDay);
6720            table.getColumnModel().addColumn(tcPlays);
6721            table.getColumnModel().addColumn(tcAge);
6722
6723            table.getColumnModel().moveColumn(6, 1);
6724            table.getColumnModel().moveColumn(6, 2);
6725            table.getColumnModel().moveColumn(6, 3);
6726         }
6727
6728         trace("setColumnVisibility()", EXIT);
```

```
6729      }
6730
6731      public void trace(String strText)
6732      {
6733         trace(strText, COMMENT);
6734      }
6735
6736      public void trace(String strText, int iType)
6737      {
6738         if (iType == ENTER)
6739         {
6740            strTrcHdr = strTrcHdr + "  ";
6741
6742            System.out.println(strTrcHdr + "+" + strText);
6743         }
6744         else
6745         {
6746            if (iType == EXIT)
6747            {
6748               System.out.println(strTrcHdr + "-" + strText);
6749
6750               if (strTrcHdr.length() >= 2)
6751                  strTrcHdr = strTrcHdr.substring(0, strTrcHdr.length() - 2);
6752            }
6753            else
6754            {
6755               System.out.println(strTrcHdr + "  " + strText);
6756            }
6757         }
6758      }
6759
6760      static public void main(String args[])
6761      {
6762         int         credits = 0;
6763         boolean     debug = false;
6764         MP3Jukeboxx mp3Jukeboxx = null;
6765
6766
6767         if (args.length > 0)
6768         {
6769            try
6770            {
6771               credits = Integer.parseInt(args[0]);
6772            }
6773            catch (java.lang.NumberFormatException e)
6774            {
6775               System.out.println("Cannot parse credits, using default of 0...");
6776
6777               credits = 0;
6778            }
6779
6780            if (args.length > 1)
6781            {
6782               if (args[1].equalsIgnoreCase("true"))
6783                  debug = true;
6784            }
6785         }
6786
6787
6788         try
6789         {
6790            UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
6791         }
6792         catch (Exception e)
6793         {
6794            System.out.println("Cannot set Windows look and feel");
6795         }
6796
6797
6798         try
6799         {
6800            logFile = new File("MP3Jukeboxx.log");
6801            raLogFile = new RandomAccessFile(logFile, "rw");
6802            raLogFile.seek(raLogFile.length());
6803
6804
6805            out = new BufferedWriter(new FileWriter(logFile));
6806
6807
6808            mp3Jukeboxx = new MP3Jukeboxx("MP3Jukeboxx", credits, debug);
6809         }
6810         catch (java.io.IOException ioException)
6811         {
6812            logException(ioException, mp3Jukeboxx);
6813         }
6814         catch (Exception e)
6815         {
6816            logException(e, mp3Jukeboxx);
6817            System.exit(1);
6818         }
```

```
6819      }
6820
6821      static public void logException(Exception e)
6822      {
6823         logException(e, null);
6824      }
6825
6826      static public void logException(Exception e, MP3Jukeboxx mp3Jukeboxx)
6827      {
6828         e.printStackTrace();
6829
6830         logInfo(e.toString());
6831
6832         if (mp3Jukeboxx != null)
6833         {
6834            mp3Jukeboxx.logAppExceptionInfo();
6835         }
6836      }
6837
6838      static public void logInfo(String str)
6839      {
6840         System.out.println(strTrcHdr + "  " + str);
6841
6842         //           1         2
6843         // 01234567890123456789012345678 9
6844         // Wed Jun 21 15:52:16 EDT 2000
6845         Date now = new Date();
6846
6847         String strLine = now.toString().substring(11,19) + ": " + str + "\r" + "\n";
6848
6849         try
6850         {
6851            if (logFile.length() >= 1048576)
6852            {
6853               logFile.delete();
6854            }
6855
6856
6857            raLogFile.writeBytes(strLine);
6858
6859
6860            /*
6861            out.write(strLine, 0, strLine.length());
6862            out.newLine();
6863            out.flush();
6864            */
6865         }
6866         catch (java.io.IOException ioException)
6867         {
6868            System.out.println("ERROR:  Could not write data to disk!");
6869         }
6870      }
6871   }
6872
```

ATTACHMENT 2

```
/**
 * Filename: TreeMgr.java
 *
 * Author: Tom Myers
 *
 * Version: 1.0
 *
 * Purpose: This file contains the code for the TreeMgr object class, used to
 *          to handle the data model associated with the JTree screen component in the
 *          main application.  This class constructs the data model by recursively parsing
 *          the given drives, creating a tree structure that maps that of the MP3s/Wavs on
 *          disk.  In addition, for each MP3/Wav file found, a PlayListEntry helper object
 *          is created and attached to the DefaultMutableTreeNode associated for that file.
 *          The PlayListEntry inner class represents the "currency" of this application, in
 *          that PlayListEntry objects populate many of the vectors and data models associated
 *          with this application.  Because the tree model as a whole is serialized and there-
 *          fore persistent, the PlayListEntry objects can maintain usage and age information
 *          each song, as well as the queued counts.
 *
 * Inputs: The following parameters are used to construct objects of this class:
 *          1. bNumberCDs - A boolean that toggles whether or not CD node objects are numbered
 *             (this will always be true for the commercial jukebox version)
 *          2. bDebug - A boolean that toggles debug information that is sent to System.out
 *
 *
 * Outputs: 1. MP3Jukeboxx.DAT - A file containing a serialized version of the DefaultTreeModel
 *          object used to populate the JTree screen component.
 *
 * Development Environment:  JDK 1.3 (Sun's Java 2 Standard Edition version 1.3) was used.
 *
 *
 * (c) Copyright 2000 Digital Jukebox Technologies LLC.  All Rights Reserved.
 */

import javax.swing.*;
import javax.swing.tree.*;
import javax.swing.ImageIcon;
import javax.swing.Icon;
import java.awt.event.*;
import java.util.Vector;
import java.util.Enumeration;
import java.util.Random;
import java.util.Calendar;
import java.util.StringTokenizer;
import java.io.*;

public class TreeMgr extends Object implements Serializable
{
   public class PlayListEntry implements Serializable
   {
      public PlayListEntry()
      {
         _mp3Path = null;
         _mp3NameIdx = 0;
         _artistIndex  = 0;
         _playedCnt = 0;
         _queuedCnt = 0;
         _paidCnt       = 0;
         _paidQueuedCnt = 0;
         _ageInDays = 0;
      }

      public PlayListEntry(String mp3Path, int iArtistIndex)
      {
         _mp3Path = mp3Path;

         _mp3NameIdx = 0;

         _artistIndex  = (short)iArtistIndex;

         _playedCnt = 0;
         _queuedCnt = 0;

         _paidCnt       = 0;
         _paidQueuedCnt = 0;

         _ageInDays = (short)TreeMgr.calculateCurrentAge();
      }

      public void  setNameIdx(byte idx) { _mp3NameIdx = idx; }
      public byte  getNameIdx() { return _mp3NameIdx; }

      public String  getCurrPlayingSong() { return TreeMgr.strCurrentSong; }
      public int     getAge() { return TreeMgr.calculateCurrentAge() - _ageInDays; }

      public void    incrementPlayedCnt() { this._playedCnt = ((short)(this._playedCnt + 1)); }
      public int     getPlayedCnt() { return this._playedCnt; }
      public void    resetPlayedCnt() { this._playedCnt = 0; }

      public void    incrementQueuedCnt() { this._queuedCnt = ((byte)(this._queuedCnt + 1)); }
```

```
91          public void    decrementQueuedCnt()
92          {
93             if (((byte)(this._queuedCnt)) > 0)
94                this._queuedCnt = ((byte)(this._queuedCnt - 1));
95             else
96                this._queuedCnt = ((byte)0);
97          }
98          public int     getQueuedCnt() { return this._queuedCnt; }
99          public void    resetQueuedCnt() { this._queuedCnt = 0; }
100
101         public void    incrementPaidCnt() { this._paidCnt = ((byte)(this._paidCnt + 1)); }
102         public int     getPaidCnt() { return this._paidCnt; }
103         public void    resetPaidCnt() { this._paidCnt = 0; }
104
105         public void    incrementPaidQueuedCnt() { this._paidQueuedCnt = ((byte)(this._paidQueuedCnt + 1)); }
106         public void    decrementPaidQueuedCnt()
107         {
108            if (((byte)(this._paidQueuedCnt)) > 0)
109               this._paidQueuedCnt = ((byte)(this._paidQueuedCnt - 1));
110            else
111               this._paidQueuedCnt = ((byte)0);
112         }
113         public int     getPaidQueuedCnt() { return this._paidQueuedCnt; }
114         public void    resetPaidQueuedCnt() { this._paidQueuedCnt = 0; }
115
116         public String  getMp3Path() { return this._mp3Path; }
117         public int     getArtistIndex() { return this._artistIndex; }
118         public void    setArtistIndex(int index) { this._artistIndex = (short)index; }
119         public String  toString()
120         {
121            int iSlashIndex = _mp3Path.lastIndexOf("\\");
122
123            return this._mp3Path.substring(iSlashIndex+1);
124         }
125
126         public int getTrackDashIndex()
127         {
128            String  mp3Name    = this.toString();
129            int     iDashIndex = mp3Name.indexOf('-');
130
131            boolean bDone = false;
132            while (bDone == false)
133            {
134               if (mp3Name.length() > iDashIndex+3)
135               {
136                  if (mp3Name.charAt(iDashIndex+3) == '-')
137                  {
138                     try
139                     {
140                        Integer.parseInt(mp3Name.substring(iDashIndex+1,iDashIndex+3));
141                        bDone = true;
142                     }
143                     catch (java.lang.NumberFormatException e)
144                     {
145                        iDashIndex  = mp3Name.indexOf('-', iDashIndex+1);
146
147                        if (iDashIndex == -1)
148                        {
149                           bDone = true;
150                        }
151                     }
152                  }
153                  else
154                  {
155                     iDashIndex  = mp3Name.indexOf('-', iDashIndex+1);
156
157                     if (iDashIndex == -1)
158                     {
159                        bDone = true;
160                     }
161                  }
162               }
163               else
164               {
165                  bDone = true;
166               }
167            }
168            return iDashIndex;
169         }
170
171         public int getTrackNum()
172         {
173            String  mp3Name     = this.toString();
174            int     iTrackNum   = -1;
175            int     iDashIndex  = getTrackDashIndex();
176            String  strTrackNum = mp3Name.substring(iDashIndex + 1, iDashIndex + 3);
177
178            try
179            {
180               iTrackNum = Integer.parseInt(strTrackNum);
```

```
            }
            catch (java.lang.NumberFormatException e)
            {
               iTrackNum = -1;
            }

            return iTrackNum;
         }

         public String getArtist()
         {
            String mp3Name    = this.toString();
            String currArtist = null;
            int    iDashIndex = getTrackDashIndex();

            if (iDashIndex > 0)
               currArtist = mp3Name.substring(0, iDashIndex);
            else
               currArtist = mp3Name;

            return currArtist;
         }

         public String getSong()
         {
            String mp3Name    = this.toString();
            String currSong   = null;
            int    iDashIndex = getTrackDashIndex();

            if (iDashIndex > 0 && mp3Name.indexOf(".mp3") > 0)
               currSong = mp3Name.substring(iDashIndex+3, mp3Name.indexOf(".mp3"));
            else
               currSong = mp3Name;

            return currSong;
         }

         public String getGenre()
         {
            String currGenre  = null;
            int    iFirstDash = _mp3Path.indexOf("\\", 3);

            if (iFirstDash != -1 && _mp3Path.length() > 3)
               currGenre = _mp3Path.substring(3, iFirstDash);
            else
               currGenre = "unknown";

            return currGenre;
         }

         public String getCDTitle()
         {
            String  currCDTitle = null;
            int     iFirstDash  = _mp3Path.indexOf("\\", 3);
            int     iNextDash   = 0;
            int     iPrevDash   = 0;
            boolean bFound      = false;

            while (iNextDash != -1)
            {
               iNextDash = _mp3Path.indexOf("\\", iFirstDash + 1);

               if (iNextDash == -1)
               {
                  currCDTitle = _mp3Path.substring(iPrevDash+1, iFirstDash);
                  bFound = true;
               }
               else
               {
                  iPrevDash = iFirstDash;
                  iFirstDash = iNextDash;
               }
            }

            if (bFound == false)
               currCDTitle = "unknown";

            return currCDTitle;
         }

         private void setQueuedCnt(int iQueuedCnt) { this._queuedCnt = ((byte)iQueuedCnt); }
         private void setPaidQueuedCnt(int iPaidQueuedCnt) { this._paidQueuedCnt = ((byte)iPaidQueuedCnt); }

         private void setPaidCnt(int iPaidCnt) { this._paidCnt = ((short)iPaidCnt); }
         private int  getRawAge() { return ((int)this._ageInDays); }
         private void setRawAge(int iAgeInDays) { this._ageInDays = ((short)iAgeInDays); }
         private void setPlayedCnt(int iPlayedCnt) { this._playedCnt = ((short)iPlayedCnt); }

         private String  _mp3Path;
         private byte    _mp3NameIdx;
```

```
271         private short   _artistIndex;
272         private short   _playedCnt;
273         private byte    _queuedCnt;
274         private short   _paidCnt;
275         private byte    _paidQueuedCnt;
276         private short   _ageInDays;     // Since 1/1/2000.
277      }
278
279      public final static int BY_ARTIST   = 0;
280      public final static int BY_CDTITLE  = 1;
281      public final static int BY_SONG     = 2;
282      public final static int BY_ALL      = 3;
283      public final static int BOOLEAN_AND = 4;
284      public final static int BOOLEAN_OR  = 5;
285
286
287      private int _iArtistCount = 0;
288      private int _iSongCount   = 0;
289
290      private DefaultTreeModel treeModel;
291
292      private static String   strCurrentSong = new String(" ");
293      private static int      currentAge = 0;
294
295      private int iLastCmd;
296      public final static int NOTHING_SELECTED = 0;
297      public final static int PLAY             = 1;
298      public final static int NEXT             = 2;
299      public final static int NEXT_BY_GENRE    = 3;
300      public final static int PREV_GENRE       = 4;
301      public final static int PREV_ARTIST      = 5;
302      public final static int PREV_BY_ARTIST   = 6;
303      public final static int NEXT_BY_ARTIST   = 7;
304      public final static int NEXT_ARTIST      = 8;
305      public final static int NEXT_GENRE       = 9;
306
307      private PlayListEntry _prevPlayListObj;
308      private PlayListEntry _currPlayListObj;
309      private PlayListEntry _nextPlayListObj;
310
311      private Random  randomNbrObj;
312      private boolean _bRandomMode;
313      public void     setRandomMode(boolean bMode) { this._bRandomMode = bMode; }
314      public boolean  getRandomMode() { return this._bRandomMode; }
315
316      private int _iRowCount;
317      public void setRowCount(int cnt) {_iRowCount = cnt; }
318      public int  getRowCount() {return _iRowCount; }
319
320      private int _iCDCount;
321      private boolean _bNumberCDs;
322      private boolean _bDebug;
323
324
325      public TreeMgr()
326      {
327         this(false, false);
328      }
329
330      public TreeMgr(boolean bNumberCDs, boolean bDebug)
331      {
332         _iArtistCount = 0;
333         _iSongCount   = 0;
334         _iRowCount    = 0;
335         _iCDCount     = 0;
336         _bNumberCDs   = bNumberCDs;
337         _bDebug       = bDebug;
338         _bRandomMode  = true;
339         randomNbrObj  = new Random( java.lang.System.currentTimeMillis() );
340
341         iLastCmd = NOTHING_SELECTED;
342      }
343
344      public PlayListEntry createBlankPlayListEntry()
345      {
346         return new PlayListEntry();
347      }
348
349      public static int calculateCurrentAge()
350      {
351         // Get the current year and day of the year from the Calendar class.
352         int iDay   = Calendar.getInstance().get(Calendar.DAY_OF_YEAR);
353         int iYear  = Calendar.getInstance().get(Calendar.YEAR);
354
355         // Foil any attempts to break this program by tampering with the system clock.
356         if ( (iYear < 2000) || (iYear >= 2100))
357         {
358            iYear = 2000;
359         }
360
```

```
361         // Calculate the number of elapsed days since our inception at 1/1/2000.
362         return (((iYear - 2000) * 365) + iDay) - 1;
363      }
364
365      public TreePath getPathForSong(String mp3)
366      {
367         DefaultMutableTreeNode root;
368         DefaultMutableTreeNode tmpNode;
369         Object                 tmpObject;
370         boolean                bDone = false;
371
372         root = (DefaultMutableTreeNode)treeModel.getRoot();
373
374         // Perform a Prefix traversal of the tree.
375         for (Enumeration e = root.preorderEnumeration(); e.hasMoreElements() && !bDone; )
376         {
377            tmpNode = (DefaultMutableTreeNode)e.nextElement();
378
379            if (isPlayListEntry(tmpNode))
380            {
381               tmpObject = tmpNode.getUserObject();
382
383               if (mp3.equalsIgnoreCase(((PlayListEntry)tmpObject).getMp3Path()))
384               {
385                  bDone = true;
386                  return new TreePath(tmpNode.getPath());
387               }
388            }
389         }
390
391         return null;
392      }
393
394      public TreePath getPathForSong(char ch, JTree tree)
395      {
396         DefaultMutableTreeNode root;
397         DefaultMutableTreeNode tmpNode;
398         Object                 tmpObject;
399         boolean                bDone = false;
400         DefaultMutableTreeNode parentNode;
401         DefaultMutableTreeNode currNode;
402
403         // Get the selected node.
404         currNode = (DefaultMutableTreeNode)tree.getLastSelectedPathComponent();
405
406         // If this is a mp3, get the parent, otherwise assume we are at a parent node already.
407         if (isPlayListEntry(currNode))
408            parentNode = (DefaultMutableTreeNode)currNode.getParent();
409         else
410            parentNode = currNode;
411
412         // Enumerate the children looking for a match.
413         for (Enumeration e = parentNode.preorderEnumeration(); e.hasMoreElements() && !bDone; )
414         {
415            tmpNode = (DefaultMutableTreeNode)e.nextElement();
416
417            if (isPlayListEntry(tmpNode))
418            {
419               tmpObject = tmpNode.getUserObject();
420
421               if (((PlayListEntry)tmpObject).toString().toLowerCase().charAt(0) == ch)
422               {
423                  bDone = true;
424                  return new TreePath(tmpNode.getPath());
425               }
426            }
427         }
428
429         return null;
430      }
431
432      public void deleteMp3(JTree tree, DefaultMutableTreeNode node, boolean bReloadTree)
433      {
434         int iRow = -1;
435         if (bReloadTree)
436            iRow = tree.getRowForPath(new TreePath(node.getPath()));
437
438         TreeNode parentNode = ((TreeNode)node).getParent();
439         ((DefaultMutableTreeNode)parentNode).remove(((MutableTreeNode)node));
440
441         treeModel.setRoot((DefaultMutableTreeNode)tree.getModel().getRoot());
442
443         // Now, display the changes.
444         reloadTree(tree, iRow);
445
446         // Update the song count.
447         _iSongCount = _iSongCount - 1;
448      }
449
450      public void reloadTree(JTree tree, int iRow)
```

```
451     {
452        ((DefaultTreeModel)tree.getModel()).reload();
453        expandTree(tree);
454
455        if (iRow != -1)
456        {
457           if ( iRow <= 10)
458           {
459              tree.setSelectionRow(1);
460              tree.scrollRowToVisible(1);
461
462              tree.setSelectionRow(iRow);
463           }
464           else
465           {
466              if ( (iRow + 10) >= tree.getRowCount())
467              {
468                 tree.setSelectionRow(1);
469                 tree.scrollRowToVisible(1);
470
471                 tree.scrollRowToVisible(tree.getRowCount() - 1);
472                 tree.setSelectionRow(iRow);
473              }
474              else
475              {
476                 tree.setSelectionRow(1);
477                 tree.scrollRowToVisible(1);
478
479                 tree.scrollRowToVisible(iRow + 10);
480                 tree.setSelectionRow(iRow);
481              }
482           }
483        }
484     }
485
486     public void expandTree(JTree tree)
487     {
488        // Expand all the nodes in the tree before displaying it.
489        int row = 0;
490        while (row <= tree.getRowCount())
491        {
492           tree.expandRow(row);
493           row++;
494        }
495     }
496
497     public void refreshTree(JTree tree)
498     {
499        DefaultMutableTreeNode root;
500        root = (DefaultMutableTreeNode)tree.getModel().getRoot();
501        refreshTree(root);
502        expandTree(tree);
503     }
504
505     public void refreshTree(DefaultMutableTreeNode root)
506     {
507        DefaultMutableTreeNode tmpNode;
508
509        // Perform a traversal of the tree.
510        for (Enumeration e = root.breadthFirstEnumeration(); e.hasMoreElements(); )
511        {
512           tmpNode = (DefaultMutableTreeNode)e.nextElement();
513
514           if (!tmpNode.isLeaf())
515           {
516              refreshDir(tmpNode);
517           }
518        }
519        treeModel.setRoot(root);
520     }
521
522     public void refreshDir(DefaultMutableTreeNode node)
523     {
524        int iPos           = -1;
525        File dir           = null;
526        File tmpFile       = null;
527        String dirList[]   = null;
528        String strName     = null;
529        String tmpString   = null;
530        TreePath treePath  = null;
531        boolean bDone      = false;
532        boolean bFoundNode = false;
533        PlayListEntry newObject        = null;
534        DefaultMutableTreeNode newNode = null;
535        DefaultMutableTreeNode tmpNode = null;
536
537
538        // Create a File object corresponding to the filesystem path for the node.
539        // Get a directory listing for this File object.
540        // Enumerate the children of the node.
```

```
541         // For each file in the directory listing, go through the enumeration.
542         // Compare the node's name to the filename (ignore directories in the listing)
543         // While the node's name is lexically less than the filename, continue the enumeration.
544         // If the node's name is equal to the filename, then quit the enumeration, result is do nothing.
545         // If the node's name is greater than the filename, then create a
546         // node object and associated PlayListEntry object, add it to the
547         // passed in node, then quit the enumeration.
548
549         treePath = new TreePath(node.getPath());
550
551
552         //            012345678901
553         // Dir path:  [Root, D:, MP3, Industrl]
554         tmpString = treePath.toString();
555
556         if (tmpString.length() >= 9)
557         {
558            tmpString = tmpString.substring(7);
559
560            iPos = tmpString.indexOf(",");
561            if (iPos != -1)
562            {
563               // 012
564               // D:]
565               strName = tmpString.substring(0, 2);
566               strName = strName.concat("\\");
567
568               //             0123456789012
569               // Node path:  D:, MP3, Industrl]
570               tmpString = tmpString.substring(4);
571
572               //01234567890123
573               //MP3, Industrl]
574               iPos = tmpString.indexOf(", ");
575               while (iPos != -1)
576               {
577                  strName = strName.concat(tmpString.substring(0, iPos));
578                  strName = strName.concat("\\");
579                  tmpString = tmpString.substring(iPos + 2);
580                  iPos = tmpString.indexOf(", ");
581               }
582
583               strName = strName.concat(tmpString);
584               strName = strName.substring(0, strName.length() - 1);
585            }
586            else
587            {
588               // 012
589               // D:]
590               strName = tmpString.substring(0, 2);
591               strName = strName.concat("\\");
592            }
593
594
595            dir = new File(strName);
596            dirList = dir.list();
597
598
599            for (int i = 0; i < dirList.length; i++)
600            {
601               tmpFile = new File(dir.getPath(), dirList[i]);
602
603
604               if (isPlayListEntry(tmpFile))
605               {
606                  bDone = false;
607                  bFoundNode = false;
608                  for (Enumeration enum = node.children(); enum.hasMoreElements() && !bDone; )
609                  {
610                     tmpNode = (DefaultMutableTreeNode)enum.nextElement();
611
612                     if (isPlayListEntry(tmpNode))
613                     {
614                        Object userObject = tmpNode.getUserObject();
615
616                        // If equal, mp3 already exists in the tree, go onto next file.
617                        if (tmpNode.toString().equalsIgnoreCase(tmpFile.getName()))
618                        {
619                           bDone = true;
620                           bFoundNode = true;
621                        }
622                     }
623                  }
624
625                  if (!bFoundNode)
626                  {
627                     newNode   = new DefaultMutableTreeNode(tmpFile.getName());
628
629                     _iSongCount   = _iSongCount + 1;
630                     _iArtistCount = _iArtistCount + 1;
```

```
631                     newObject = new PlayListEntry(tmpFile.getPath(), _iArtistCount);
632
633                     newNode.setUserObject(newObject);
634
635
636                     //Now, add the new node to the tree in alphabetical order...
637                     // Start at the first child and compare Artist names, inserting the new child at
638                     // the top if lowest, before a node if lower than it, or at the end if greater than all
639                     // void insert(MutableTreeNode newChild, int childIndex)
640
641                     boolean bDone2 = false;
642                     int iTmpIndex;
643                     DefaultMutableTreeNode cmpNode = null;
644                     Object cmpObject = null;
645
646                     for (Enumeration e = node.children(); (e.hasMoreElements() && !bDone2) ; )
647                     {
648                        cmpNode   = (DefaultMutableTreeNode)e.nextElement();
649                        cmpObject = cmpNode.getUserObject();
650
651                        if (cmpObject != null && cmpObject instanceof PlayListEntry)
652                        {
653                           if (newObject.getArtist().compareTo(((PlayListEntry)cmpObject).getArtist()) == 0)
654                           {
655                              newObject.setArtistIndex(((PlayListEntry)cmpObject).getArtistIndex());
656                              iTmpIndex = node.getIndex(cmpNode);
657                              node.insert(newNode, iTmpIndex++);
658                              bDone2 = true;
659                           }
660                           else
661                           {
662                              if (newObject.getArtist().compareTo(((PlayListEntry)cmpObject).getArtist()) < 0)
663                              {
664                                 //need to bump up all artist index counts {ugh}
665                                 iTmpIndex = node.getIndex(cmpNode);
666                                 node.insert(newNode, iTmpIndex--);
667                                 bDone2 = true;
668                              }
669                           }
670                        }
671                     }
672
673                     // The current artist is lexically greater than all the other artists.
674                     if (bDone2 == false)
675                     {
676                        newObject.setArtistIndex(((PlayListEntry)cmpObject).getArtistIndex() + 1);
677                        node.add(newNode);
678                     }
679                  }
680               }
681
682               // If a directory, see if the directory already exists in the tree.
683               if (tmpFile.isDirectory())
684               {
685                  bDone = false;
686                  bFoundNode = false;
687                  for (Enumeration enum = node.children(); enum.hasMoreElements() && !bDone; )
688                  {
689                     tmpNode = (DefaultMutableTreeNode)enum.nextElement();
690
691                     if (tmpNode.toString().indexOf('-') == 3 && tmpNode.toString().length() > 3)
692                     {
693                        if (tmpNode.toString().substring(4).equalsIgnoreCase(tmpFile.getName()))
694                        {
695                           bDone = true;
696                           bFoundNode = true;
697                        }
698                     }
699                     else
700                     {
701                        if (tmpNode.toString().equalsIgnoreCase(tmpFile.getName()))
702                        {
703                           bDone = true;
704                           bFoundNode = true;
705                        }
706                     }
707                  }
708
709                  if (!bFoundNode)
710                  {
711                     DefaultMutableTreeNode addNode = recurseDir(tmpFile);
712
713                     tmpNode.add(addNode);
714                  }
715               }
716            }
717         }
718      }
719
720      public DefaultMutableTreeNode getNodeForSong(String mp3)
```

```
721     {
722        DefaultMutableTreeNode root       = (DefaultMutableTreeNode)treeModel.getRoot();
723        DefaultMutableTreeNode tmpNode    = null;
724        Object                 userObject = null;
725        boolean                bDone      = false;
726
727        // Perform a Prefix traversal of the tree.
728        for (Enumeration e = root.preorderEnumeration(); e.hasMoreElements() && !bDone; )
729        {
730           tmpNode = (DefaultMutableTreeNode)e.nextElement();
731           userObject = tmpNode.getUserObject();
732
733           if (userObject != null && userObject instanceof PlayListEntry)
734           {
735              if (mp3.equalsIgnoreCase(((PlayListEntry)userObject).getMp3Path()))
736              {
737                 bDone = true;
738              }
739           }
740        }
741
742        return tmpNode;
743     }
744
745
746     public PlayListEntry getPlayListObjForSong(String mp3)
747     {
748        DefaultMutableTreeNode root;
749        DefaultMutableTreeNode tmpNode;
750        PlayListEntry          tmpObject = null;
751        boolean                bDone = false;
752
753        root = (DefaultMutableTreeNode)treeModel.getRoot();
754
755        // Perform a Prefix traversal of the tree.
756        for (Enumeration e = root.preorderEnumeration(); e.hasMoreElements() && !bDone; )
757        {
758           tmpNode = (DefaultMutableTreeNode)e.nextElement();
759
760           if (isPlayListEntry(tmpNode))
761           {
762              tmpObject = (PlayListEntry)tmpNode.getUserObject();
763
764              if (mp3.equalsIgnoreCase(tmpObject.getMp3Path()))
765              {
766                 bDone = true;
767                 return tmpObject;
768              }
769           }
770        }
771        return null;
772     }
773
774     public int getLastCmd()
775     {
776        return iLastCmd;
777     }
778
779     public void setSelectedMp3BySong(String mp3, JTree tree)
780     {
781        TreePath treePath = getPathForSong(mp3);
782
783        if (treePath == null)
784        {
785           System.out.println("Error selecting song: " + mp3);
786        }
787        else
788        {
789           int iRow = tree.getRowForPath(treePath);
790
791           if ( iRow <= 10)
792           {
793              tree.setSelectionRow(1);
794              tree.scrollRowToVisible(1);
795
796              tree.setSelectionRow(iRow);
797           }
798           else
799           {
800              if ((iRow + 10) >= tree.getRowCount())
801              {
802                 tree.setSelectionRow(1);
803                 tree.scrollRowToVisible(1);
804
805                 tree.scrollRowToVisible(tree.getRowCount() - 1);
806                 tree.setSelectionRow(iRow);
807              }
808              else
809              {
810                 tree.setSelectionRow(1);
```

```
811                tree.scrollRowToVisible(1);
812
813                tree.scrollRowToVisible(iRow + 10);
814                tree.setSelectionRow(iRow);
815             }
816          }
817       }
818    }
819
820    public void setSelectedMp3BySong(char ch, JTree tree)
821    {
822       TreePath treePath = getPathForSong(ch, tree);
823
824       if (treePath != null)
825       {
826          int iRow = tree.getRowForPath(treePath);
827
828          if ( iRow <= 10)
829          {
830             tree.setSelectionRow(1);
831             tree.scrollRowToVisible(1);
832
833             tree.setSelectionRow(iRow);
834          }
835          else
836          {
837             if ((iRow + 10) >= tree.getRowCount())
838             {
839                tree.setSelectionRow(1);
840                tree.scrollRowToVisible(1);
841
842                tree.scrollRowToVisible(tree.getRowCount() - 1);
843                tree.setSelectionRow(iRow);
844             }
845             else
846             {
847                tree.setSelectionRow(1);
848                tree.scrollRowToVisible(1);
849
850                tree.scrollRowToVisible(iRow + 10);
851                tree.setSelectionRow(iRow);
852             }
853          }
854       }
855    }
856
857
858    public Object selectedMp3(JTree tree)
859    {
860        Object selectedNode = tree.getLastSelectedPathComponent();
861        Object userObject   = null;
862
863        if (isPlayListEntry((DefaultMutableTreeNode)selectedNode))
864        {
865           userObject = ((DefaultMutableTreeNode)selectedNode).getUserObject();
866        }
867
868        return userObject;
869    }
870
871    public PlayListEntry next(JTree tree)
872    {
873       iLastCmd = NEXT;
874       boolean bDone = false;
875
876       // If we hit 100 songs already played, then force a selection.
877       int iAlreadyPlayedCnt = 0;
878       Object userObject = null;
879       Object selectedNode = null;
880       DefaultMutableTreeNode nextNode = new DefaultMutableTreeNode();
881
882       //If in Random mode, pick song not already played (at random, obviously), then make it the selected song.
883       if (_bRandomMode)
884       {
885          while (bDone != true)
886          {
887             userObject = null;
888
889             int iRnd = randomNbrObj.nextInt();
890             iRnd = Math.abs(iRnd);
891             iRnd = iRnd % _iRowCount;
892             iRnd += 1;
893
894             tree.setSelectionRow(iRnd);
895
896             selectedNode = tree.getLastSelectedPathComponent();
897             if (selectedNode instanceof DefaultMutableTreeNode)
898             {
899                userObject = ((DefaultMutableTreeNode)selectedNode).getUserObject();
900
```

```
901                     if (userObject instanceof PlayListEntry)
902                     {
903                        if ((((PlayListEntry)userObject).getPlayedCnt() == 0) && (((PlayListEntry)userObject).
getQueuedCnt() == 0))
904                        {
905                           nextNode = (DefaultMutableTreeNode)selectedNode;
906                           bDone = true;
907                        }
908                        else
909                        {
910                           iAlreadyPlayedCnt = iAlreadyPlayedCnt + 1;
911
912                           if (iAlreadyPlayedCnt == 100)
913                           {
914                              bDone = true;
915                              nextNode = (DefaultMutableTreeNode)selectedNode;
916                           }
917                        }
918                     }
919                  }
920               }
921               TreePath treePath = new TreePath(nextNode.getPath());
922               tree.setSelectionPath(treePath);
923            }
924            else
925            {
926               // Get the currently selected mp3.
927               nextNode = (DefaultMutableTreeNode)tree.getLastSelectedPathComponent();
928               if (nextNode != null)
929                  userObject = nextNode.getUserObject();
930               else
931                  userObject = null;
932
933               // Make sure we get an mp3 to start with.
934               if (userObject == null || !(userObject instanceof PlayListEntry))
935               {
936                  bDone = false;
937
938                  while (!bDone)
939                  {
940                     // In case nothing is selected, select a song at random now...
941                     int iRnd = randomNbrObj.nextInt();
942                     iRnd = Math.abs(iRnd);
943                     iRnd = iRnd % _iRowCount;
944                     iRnd += 1;
945
946                     tree.setSelectionRow(iRnd);
947
948                     nextNode = (DefaultMutableTreeNode)tree.getLastSelectedPathComponent();
949
950                     if (nextNode != null)
951                        userObject = nextNode.getUserObject();
952
953                     if (userObject != null && userObject instanceof PlayListEntry)
954                        bDone = true;
955                  }
956               }
957
958
959               bDone = false;
960               while (!bDone)
961               {
962                  // Get the next, in sequential order, mp3 that hasn't been played.
963                  nextNode = ((DefaultMutableTreeNode)nextNode).getNextSibling();
964                  if (nextNode != null)
965                     userObject = nextNode.getUserObject();
966
967                  if (userObject != null && userObject instanceof PlayListEntry)
968                  {
969                     if ((((PlayListEntry)userObject).getPlayedCnt() == 0 && (((PlayListEntry)userObject).getQueuedCnt
() == 0)))
970                        bDone = true;
971                     else
972                     {
973                        iAlreadyPlayedCnt = iAlreadyPlayedCnt + 1;
974
975                        if (iAlreadyPlayedCnt == 100)
976                           bDone = true;
977                     }
978                  }
979               }
980
981               TreePath treePath = new TreePath(nextNode.getPath());
982               tree.setSelectionPath(treePath);
983               tree.scrollRowToVisible(tree.getRowForPath(treePath));
984
985            }
986
987            PlayListEntry mp3 = (PlayListEntry)nextNode.getUserObject();
988
```

```
989          return (mp3);
990       }
991
992       public PlayListEntry nextByGenre(JTree tree)
993       {
994          iLastCmd = NEXT_BY_GENRE;
995
996          boolean bDone = false;
997
998          DefaultMutableTreeNode currNode = (DefaultMutableTreeNode)tree.getLastSelectedPathComponent();
999          DefaultMutableTreeNode parentNode = (DefaultMutableTreeNode)currNode.getParent();
1000         DefaultMutableTreeNode nextNode = new DefaultMutableTreeNode();
1001         PlayListEntry nextObject;
1002         int iGenreCount = parentNode.getChildCount();
1003         TreePath treePath;
1004
1005         if (_bRandomMode)
1006         {
1007            while (bDone != true)
1008            {
1009               int iRnd = randomNbrObj.nextInt();
1010               iRnd = Math.abs(iRnd);
1011               iRnd = iRnd % iGenreCount;
1012
1013               nextNode = (DefaultMutableTreeNode)treeModel.getChild(parentNode, iRnd);
1014               nextObject = (PlayListEntry)nextNode.getUserObject();
1015
1016               if (nextObject.getPlayedCnt() == 0 && nextObject.getQueuedCnt() == 0)
1017               {
1018                  _currPlayListObj = nextObject;
1019                  bDone = true;
1020               }
1021
1022               treePath = new TreePath(nextNode.getPath());
1023
1024               tree.setSelectionPath(treePath);
1025               tree.scrollPathToVisible(treePath);
1026            }
1027         }
1028
1029         return ((PlayListEntry)nextNode.getUserObject());
1030      }
1031
1032      public void skip(JTree tree, boolean bPlayedFlag)
1033      {
1034         DefaultMutableTreeNode currNode = (DefaultMutableTreeNode)tree.getLastSelectedPathComponent();
1035         PlayListEntry userObject = (PlayListEntry)currNode.getUserObject();
1036
1037         if (bPlayedFlag == true)
1038            userObject.incrementPlayedCnt();
1039
1040         userObject.decrementQueuedCnt();
1041      }
1042
1043      public void selectInitialSong(JTree tree)
1044      {
1045         DefaultMutableTreeNode root = (DefaultMutableTreeNode)tree.getModel().getRoot();
1046         DefaultMutableTreeNode node = (DefaultMutableTreeNode)root.getFirstLeaf();
1047
1048         TreePath treePath = new TreePath(node.getPath());
1049         tree.scrollRowToVisible(1);
1050         tree.setSelectionPath(treePath);
1051      }
1052
1053      public PlayListEntry prevGenre(JTree tree)
1054      {
1055          iLastCmd = PREV_GENRE;
1056
1057          DefaultMutableTreeNode nextNode = null;
1058          DefaultMutableTreeNode currNode = (DefaultMutableTreeNode)tree.getLastSelectedPathComponent();
1059          DefaultMutableTreeNode parentNode = (DefaultMutableTreeNode)currNode.getParent();
1060          DefaultMutableTreeNode nextGenreNode = parentNode.getPreviousSibling();
1061
1062          if (nextGenreNode == null)
1063          {
1064             DefaultMutableTreeNode grandParentNode = (DefaultMutableTreeNode)parentNode.getParent();
1065             nextGenreNode = (DefaultMutableTreeNode)grandParentNode.getLastChild();
1066
1067             nextNode = (DefaultMutableTreeNode)nextGenreNode.getFirstLeaf();
1068          }
1069          else
1070          {
1071             nextNode = (DefaultMutableTreeNode)nextGenreNode.getFirstLeaf();
1072          }
1073
1074          TreePath treePath = new TreePath(nextNode.getPath());
1075          tree.setSelectionPath(treePath);
1076          tree.scrollRowToVisible(tree.getRowForPath(treePath));
1077
1078
```

```
1080            return ((PlayListEntry)nextNode.getUserObject());
1081
1082        }
1083
1084        public PlayListEntry prevArtist(JTree tree)
1085        {
1086            iLastCmd = PREV_ARTIST;
1087
1088            boolean bDone = false;
1089            DefaultMutableTreeNode currNode = (DefaultMutableTreeNode)tree.getLastSelectedPathComponent();
1090            Object currObject = ((DefaultMutableTreeNode)currNode).getUserObject();
1091            DefaultMutableTreeNode prevNode = currNode.getPreviousSibling();
1092
1093            if (prevNode == null) {
1094               prevNode = currNode;
1095            }
1096            else
1097            {
1098               Object prevObject = ((DefaultMutableTreeNode)prevNode).getUserObject();
1099
1100               if (((PlayListEntry)prevObject).getArtistIndex() != ((PlayListEntry)currObject).getArtistIndex())
1101               {
1102                   //System.out.println("Previous mp3 is by different artist...");
1103               }
1104               else
1105               {
1106                   while (bDone != true)
1107                   {
1108                       if (((PlayListEntry)prevObject).getArtistIndex() != ((PlayListEntry)currObject).getArtistIndex
()
1109                       {
1110                           bDone = true;
1111                       }
1112                       else
1113                       {
1114                           currNode = prevNode;
1115                           currObject = ((DefaultMutableTreeNode)currNode).getUserObject();
1116
1117                           prevNode = currNode.getPreviousSibling();
1118                           prevObject = ((DefaultMutableTreeNode)prevNode).getUserObject();
1119                       }
1120                   }
1121               }
1122
1123               //Now, find the first song by the previous artist.
1124               bDone = false;
1125
1126               currNode = prevNode;
1127               currObject = ((DefaultMutableTreeNode)currNode).getUserObject();
1128
1129               prevNode = currNode.getPreviousSibling();
1130               prevObject = ((DefaultMutableTreeNode)prevNode).getUserObject();
1131
1132               while (bDone != true)
1133               {
1134                   if (((PlayListEntry)prevObject).getArtistIndex() != ((PlayListEntry)currObject).getArtistIndex())
1135                   {
1136                       TreePath treePath = new TreePath(currNode.getPath());
1137                       tree.setSelectionPath(treePath);
1138                       tree.scrollRowToVisible(tree.getRowForPath(treePath));
1139
1140                       prevNode = currNode;
1141                       bDone = true;
1142                   }
1143                   else
1144                   {
1145                       currNode = prevNode;
1146                       currObject = ((DefaultMutableTreeNode)currNode).getUserObject();
1147
1148                       prevNode = currNode.getPreviousSibling();
1149                       prevObject = ((DefaultMutableTreeNode)prevNode).getUserObject();
1150                   }
1151               }
1152
1153
1154           }
1155
1156           return ((PlayListEntry)prevNode.getUserObject());
1157       }
1158
1159       public PlayListEntry prevByArtist(JTree tree)
1160       {
1161           iLastCmd = PREV_BY_ARTIST;
1162
1163           DefaultMutableTreeNode currNode = (DefaultMutableTreeNode)tree.getLastSelectedPathComponent();
1164           Object currObject = ((DefaultMutableTreeNode)currNode).getUserObject();
1165           DefaultMutableTreeNode prevNode = currNode.getPreviousSibling();
1166           Object prevObject = ((DefaultMutableTreeNode)prevNode).getUserObject();
1167
1168
```

```
1169            if (((PlayListEntry)prevObject).getArtistIndex() == ((PlayListEntry)currObject).getArtistIndex())
1170            {
1171                TreePath treePath = new TreePath(prevNode.getPath());
1172                tree.setSelectionPath(treePath);
1173                tree.scrollRowToVisible(tree.getRowForPath(treePath));
1174            }
1175            else
1176            {
1177                boolean bDone = false;
1178                DefaultMutableTreeNode nextNode = currNode.getNextSibling();
1179                Object nextObject = ((DefaultMutableTreeNode)nextNode).getUserObject();
1180
1181                while (bDone != true)
1182                {
1183                    if (((PlayListEntry)nextObject).getArtistIndex() != ((PlayListEntry)currObject).getArtistIndex())
1184                    {
1185                        TreePath treePath = new TreePath(currNode.getPath());
1186                        tree.setSelectionPath(treePath);
1187                        tree.scrollRowToVisible(tree.getRowForPath(treePath));
1188
1189                        prevNode = currNode;
1190                        bDone = true;
1191                    }
1192                    else
1193                    {
1194                        currNode = nextNode;
1195                        currObject = ((DefaultMutableTreeNode)currNode).getUserObject();
1196
1197                        nextNode = currNode.getNextSibling();
1198                        nextObject = ((DefaultMutableTreeNode)nextNode).getUserObject();
1199                    }
1200                }
1201            }
1202
1203
1204
1205            return ((PlayListEntry)prevNode.getUserObject());
1206        }
1207
1208        public PlayListEntry prevByArtistBySong(String mp3, JTree tree)
1209        {
1210           TreePath treePath = getPathForSong(mp3);
1211           tree.setSelectionPath(treePath);
1212
1213           return prevByArtist(tree);
1214        }
1215
1216        public PlayListEntry nextByArtistBySong(String mp3, JTree tree)
1217        {
1218           TreePath treePath = getPathForSong(mp3);
1219           tree.setSelectionPath(treePath);
1220
1221           return nextByArtist(tree);
1222        }
1223
1224        public PlayListEntry nextByArtist(JTree tree)
1225        {
1226            iLastCmd = NEXT_BY_ARTIST;
1227
1228            DefaultMutableTreeNode currNode = (DefaultMutableTreeNode)tree.getLastSelectedPathComponent();
1229            Object currObject = ((DefaultMutableTreeNode)currNode).getUserObject();
1230            DefaultMutableTreeNode nextNode = currNode.getNextSibling();
1231            Object nextObject = ((DefaultMutableTreeNode)nextNode).getUserObject();
1232
1233            if (((PlayListEntry)nextObject).getArtistIndex() == ((PlayListEntry)currObject).getArtistIndex())
1234            {
1235                TreePath treePath = new TreePath(nextNode.getPath());
1236                tree.setSelectionPath(treePath);
1237                tree.scrollRowToVisible(tree.getRowForPath(treePath));
1238            }
1239            else
1240            {
1241                boolean bDone = false;
1242                DefaultMutableTreeNode prevNode = currNode.getPreviousSibling();
1243                Object prevObject = ((DefaultMutableTreeNode)prevNode).getUserObject();
1244
1245                while (bDone != true)
1246                {
1247                    if (((PlayListEntry)prevObject).getArtistIndex() != ((PlayListEntry)currObject).getArtistIndex())
1248                    {
1249                        //System.out.println("Found the first song by the current artist.");
1250
1251                        TreePath treePath = new TreePath(currNode.getPath());
1252                        tree.setSelectionPath(treePath);
1253                        tree.scrollRowToVisible(tree.getRowForPath(treePath));
1254
1255                        nextNode = currNode;
1256                        bDone = true;
1257                    }
1258                    else
```

```
                {
                    //System.out.println("Finding the previous sibling to the current artist...");

                    currNode = prevNode;
                    currObject = ((DefaultMutableTreeNode)currNode).getUserObject();

                    prevNode = currNode.getPreviousSibling();
                    prevObject = ((DefaultMutableTreeNode)prevNode).getUserObject();
                }
            }
        }



        return ((PlayListEntry)nextNode.getUserObject());
    }

    public PlayListEntry nextArtist(JTree tree)
    {
        iLastCmd = NEXT_ARTIST;

        boolean bDone = false;
        DefaultMutableTreeNode currNode = (DefaultMutableTreeNode)tree.getLastSelectedPathComponent();
        Object currObject = ((DefaultMutableTreeNode)currNode).getUserObject();
        DefaultMutableTreeNode nextNode = currNode.getNextSibling();

        if (nextNode == null)
        {
           nextNode = currNode;
        }
        else
        {
           Object nextObject = ((DefaultMutableTreeNode)nextNode).getUserObject();

           if (((PlayListEntry)nextObject).getArtistIndex() != ((PlayListEntry)currObject).getArtistIndex())
           {
               TreePath treePath = new TreePath(nextNode.getPath());
               tree.setSelectionPath(treePath);
               tree.scrollRowToVisible(tree.getRowForPath(treePath));
           }
           else
           {
               currNode = nextNode;
               currObject = ((DefaultMutableTreeNode)currNode).getUserObject();

               nextNode = currNode.getNextSibling();
               nextObject = ((DefaultMutableTreeNode)nextNode).getUserObject();

               while (bDone != true)
               {
                   if (((PlayListEntry)nextObject).getArtistIndex() != ((PlayListEntry)currObject).getArtistIndex
()))
                   {
                       TreePath treePath = new TreePath(nextNode.getPath());
                       tree.setSelectionPath(treePath);
                       tree.scrollRowToVisible(tree.getRowForPath(treePath));

                       bDone = true;
                   }
                   else
                   {
                       currNode = nextNode;
                       currObject = ((DefaultMutableTreeNode)currNode).getUserObject();

                       nextNode = currNode.getNextSibling();
                       nextObject = ((DefaultMutableTreeNode)nextNode).getUserObject();
                   }
               }
           }

        }

        return ((PlayListEntry)nextNode.getUserObject());
    }

    public PlayListEntry nextGenre(JTree tree)
    {
        iLastCmd = NEXT_GENRE;

        DefaultMutableTreeNode nextNode = null;
        DefaultMutableTreeNode currNode = (DefaultMutableTreeNode)tree.getLastSelectedPathComponent();
        DefaultMutableTreeNode parentNode = (DefaultMutableTreeNode)currNode.getParent();
        DefaultMutableTreeNode nextGenreNode = parentNode.getNextSibling();

        if (nextGenreNode == null)
        {
           //System.out.println("At last Genre, retrieving first one...");

           DefaultMutableTreeNode grandParentNode = (DefaultMutableTreeNode)parentNode.getParent();
           nextGenreNode = (DefaultMutableTreeNode)grandParentNode.getFirstChild();
```

```
1349            nextNode = (DefaultMutableTreeNode)nextGenreNode.getFirstLeaf();
1350         }
1351         else
1352         {
1353            nextNode = (DefaultMutableTreeNode)nextGenreNode.getFirstLeaf();
1354         }
1355
1356         TreePath treePath = new TreePath(nextNode.getPath());
1357         tree.setSelectionPath(treePath);
1358         tree.scrollRowToVisible(tree.getRowForPath(treePath));
1359
1360
1361
1362         return ((PlayListEntry)nextNode.getUserObject());
1363      }
1364
1365      public String getCurrentlyPlayingSong()
1366      {
1367         return strCurrentSong;
1368      }
1369
1370      public void setCurrentlyPlayingSong(String strCurrSong)
1371      {
1372         strCurrentSong = strCurrSong;
1373      }
1374
1375      public void setVisibleCurrentMp3(JTree tree)
1376      {
1377         Object selectedNode = tree.getLastSelectedPathComponent();
1378
1379         if (selectedNode instanceof DefaultMutableTreeNode)
1380         {
1381            TreePath treePath = new TreePath(((DefaultMutableTreeNode)selectedNode).getPath());
1382
1383            tree.scrollRowToVisible(tree.getRowForPath(treePath));
1384         }
1385         tree.repaint(tree.getVisibleRect());
1386      }
1387
1388
1389      public void addPathToTree(JTree tree, File path)
1390      {
1391         int iPos;
1392         boolean bDone       = false;
1393         boolean bFoundNode  = false;
1394         String tmpString    = null;
1395         String strRemainder = null;
1396         DefaultMutableTreeNode root = (DefaultMutableTreeNode)tree.getModel().getRoot();
1397         DefaultMutableTreeNode childNode = null;
1398         DefaultMutableTreeNode driveNode = null;
1399         DefaultMutableTreeNode tmpNode   = null;
1400         DefaultMutableTreeNode newNode   = null;
1401
1402
1403         // Need to add path to appropriate child of root, e.g. d:\mp3test will
1404         // be added to the D: node if it exists, created if not.
1405         // Then, mp3test will be put in the appropriate path.
1406
1407
1408         if (isRoot(path))
1409         {
1410            tmpString = path.toString().substring(0, 2);
1411            strRemainder = "";
1412         }
1413         else
1414         {
1415            tmpString = path.getAbsolutePath().substring(0, 2);
1416            strRemainder = path.getAbsolutePath().substring(2);
1417         }
1418
1419
1420         // Next, see if the drive has a node in the tree already.
1421         bDone = false;
1422         bFoundNode = false;
1423         for (Enumeration e = root.children(); e.hasMoreElements() && !bDone; )
1424         {
1425            tmpNode = (DefaultMutableTreeNode)e.nextElement();
1426
1427            if (tmpNode.toString().equalsIgnoreCase(tmpString))
1428            {
1429               bDone = true;
1430               bFoundNode = true;
1431            }
1432         }
1433
1434
1435         // If the drive doesn't exist in the tree, create an entry for it.
1436         if (bFoundNode == false)
1437         {
1438            if (isRoot(path))
```

```
1439            {
1440               childNode = root;
1441            }
1442            else
1443            {
1444               childNode = new DefaultMutableTreeNode(tmpString);
1445
1446               // Need to add the node alphabetically to the tree.
1447               boolean bDone2 = false;
1448               DefaultMutableTreeNode cmpNode = null;
1449
1450               for (Enumeration e = root.children(); (e.hasMoreElements() && !bDone2) ; )
1451               {
1452                  cmpNode = (DefaultMutableTreeNode)e.nextElement();
1453
1454                  if (childNode.toString().substring(0,1).compareTo(cmpNode.toString().substring(0,1)) < 0)
1455                  {
1456                     int iIndex = treeModel.getIndexOfChild(root, cmpNode);
1457                     if (iIndex > 0)
1458                        iIndex = iIndex - 1;
1459
1460
1461                     root.insert(childNode, iIndex);
1462                     bDone2 = true;
1463                  }
1464               }
1465
1466               // The current node is lexically greater than all others.
1467               if (bDone2 == false)
1468               {
1469                  root.add(childNode);
1470               }
1471            }
1472         }
1473         else
1474         {
1475            childNode = tmpNode;
1476         }
1477
1478
1479         // Next, parse through the rest of the path, creating nodes as necessary.
1480         while (strRemainder.indexOf("\\") != -1)
1481         {
1482            iPos = strRemainder.indexOf("\\");
1483            if (iPos != -1)
1484            {
1485               strRemainder = strRemainder.substring(iPos + 1);
1486
1487               iPos = strRemainder.indexOf("\\");
1488               if (iPos != -1)
1489                  tmpString = strRemainder.substring(0, iPos);
1490               else
1491                  tmpString = strRemainder;
1492
1493
1494               bDone = false;
1495               bFoundNode = false;
1496
1497               for (Enumeration e = childNode.children(); e.hasMoreElements() && !bDone; )
1498               {
1499                  tmpNode = (DefaultMutableTreeNode)e.nextElement();
1500
1501                  if (tmpNode.toString().equalsIgnoreCase(tmpString))
1502                  {
1503                     bDone = true;
1504                     bFoundNode = true;
1505                  }
1506               }
1507
1508               if (bFoundNode == false)
1509               {
1510                  newNode = new DefaultMutableTreeNode(tmpString);
1511
1512                  childNode.add(newNode);
1513
1514                  childNode = newNode;
1515               }
1516               else
1517               {
1518                  childNode = tmpNode;
1519               }
1520            }
1521         }
1522
1523
1524         // Get the location of the currently selected node.
1525         int iRow = 0;
1526         Object selectedNode;
1527         selectedNode = tree.getLastSelectedPathComponent();
1528
```

```
1529        if (selectedNode != null && selectedNode instanceof DefaultMutableTreeNode)
1530           iRow = tree.getRowForPath(new TreePath(((DefaultMutableTreeNode)selectedNode).getPath()));
1531
1532
1533        if (bFoundNode == true && isRoot(path))
1534        {
1535           dumpCDStats();
1536
1537           rollBackupDataFiles(false); // The false signifies to not roll all the previous backups,
1538                                       // just backup the current data file to d:\backup1.
1539
1540           _iArtistCount = 0;
1541           _iSongCount   = 0;
1542           _iRowCount    = 0;
1543           _iCDCount     = 0;
1544
1545           root = new DefaultMutableTreeNode("Root");
1546           treeModel = new DefaultTreeModel(root);
1547
1548           DefaultMutableTreeNode addNode = recurseDir(path);
1549
1550           root.add(addNode);
1551           tree.setModel(treeModel);
1552
1553
1554           // Now, once everything has been added to the tree, attempt to restore statistics.
1555           restoreCDStats();
1556        }
1557        else
1558        {
1559           DefaultMutableTreeNode addNode = recurseDir(path);
1560
1561
1562           // This solves the duplicate directory problem when adding a path.
1563           if (!isRoot(path))
1564           {
1565              DefaultMutableTreeNode tempNode = (DefaultMutableTreeNode)childNode.getParent();
1566              tempNode.remove(childNode);
1567              childNode = tempNode;
1568           }
1569
1570           if (scanNodeForMp3s(addNode))
1571           {
1572              // Need to add the node alphabetically to the tree.
1573              boolean bDone3 = false;
1574              DefaultMutableTreeNode compNode = null;
1575
1576              for (Enumeration e = childNode.children(); (e.hasMoreElements() && !bDone3) ; )
1577              {
1578                 compNode = (DefaultMutableTreeNode)e.nextElement();
1579
1580                 if (addNode.toString().compareTo(compNode.toString()) < 0)
1581                 {
1582                    int index = treeModel.getIndexOfChild(root, compNode);
1583
1584                    if (index > 0)
1585                       index = index - 1;
1586
1587                    childNode.insert(addNode, index);
1588                    bDone3 = true;
1589                 }
1590              }
1591
1592              // The current node is lexically greater than all others.
1593              if (bDone3 == false)
1594              {
1595                 childNode.add(addNode);
1596              }
1597           }
1598
1599
1600           // Attempt to restore the same visibility to the tree.
1601           treeModel.setRoot(root);
1602           reloadTree(tree, iRow);
1603
1604
1605           // Now, once everything has been added to the tree, attempt to restore statistics.
1606           restoreCDStats();
1607        }
1608     }
1609
1610
1611     private boolean isRoot(File dir)
1612     {
1613        int iPos = dir.toString().indexOf(":");
1614
1615        if (iPos != -1 && dir.toString().length() == 3)
1616           return true;
1617        else
1618           return false;
```

```
1619      }
1620
1621      /*
1622       * If the directoy has at least one child that is an MP3, then
1623       * assume that this directory represents a CD, and if so,
1624       * return true, otherwise, return false.
1625       */
1626      private boolean isCDNode(File dir)
1627      {
1628         boolean bIsCDNode = false;
1629         boolean bDone1    = false;
1630         boolean bDone2    = false;
1631         File    tmpFile   = null;
1632         String  dirList[] = dir.list();
1633
1634         for (int i = 0; i < dirList.length && (!bDone1 || !bDone2); i++)
1635         {
1636            tmpFile = new File(dir.getPath(), dirList[i]);
1637
1638            if (isPlayListEntry(tmpFile))
1639            {
1640               bDone1 = true;
1641            }
1642
1643            if (tmpFile.toString().toLowerCase().indexOf("cover.jpg") != -1)
1644            {
1645               bDone2 = true;
1646            }
1647         }
1648
1649         if (bDone1 == true && bDone2 == true)
1650         {
1651            bIsCDNode = true;
1652         }
1653
1654         return bIsCDNode;
1655      }
1656
1657      /** This method sets the CD Count data member such that the next time a CD or CDs are
1658       * added to an existing tree model, then those CDs will be numbered accordingly.
1659       * note:  the CD's are a zero-indexed "array" if the CD count is at 179, it means 180 total CDs.
1660       */
1661      public void setNewCDIndex(int iNewCount)
1662      {
1663         if (iNewCount > 0 )
1664         {
1665            _iCDCount = iNewCount;
1666         }
1667      }
1668
1669      /** The following recursive method takes a directory and returns the
1670       * corresponding DefaultMutableTreeNode.  Leafs are added and
1671       * must be a file with the .mp3 extension.
1672       */
1673      public DefaultMutableTreeNode recurseDir(File dir)
1674      {
1675         DefaultMutableTreeNode node     = null;
1676         String                 nodeName = null;
1677
1678         if (isRoot(dir))
1679         {
1680            nodeName = dir.toString().substring(0, 2);
1681         }
1682         else
1683         {
1684            if (_bNumberCDs == true)
1685            {
1686               // If this directory has MP3s for children, then boldly assume that this directory
1687               // represents a CD.  As such, increment the CD counter and prepend this number to
1688               // the node name.
1689               if (isCDNode(dir))
1690               {
1691                  if (_iCDCount < 10)
1692                     nodeName = "00" + Integer.toString(_iCDCount) + "-" + dir.getName();
1693                  else if (_iCDCount < 100)
1694                     nodeName = "0" + Integer.toString(_iCDCount) + "-" + dir.getName();
1695                  else
1696                     nodeName = Integer.toString(_iCDCount) + "-" + dir.getName();
1697
1698                  _iCDCount = _iCDCount + 1;
1699
1700                  // System.out.println("Setting CD Name: " + nodeName);
1701               }
1702               else
1703               {
1704                  nodeName = dir.getName();
1705               }
1706            }
1707            else
1708            {
```

```
1709                nodeName = dir.getName();
1710             }
1711          }
1712          node = new DefaultMutableTreeNode(nodeName);
1713
1714          Vector tempList = new Vector();
1715          Vector tempNode = new Vector();
1716          Vector fileList = new Vector();
1717          String prevArtist = new String("#");
1718          String currArtist = new String("");
1719          int iDashIndex;
1720          boolean bFirstChildCreated = false;
1721          DefaultMutableTreeNode firstChild;
1722          File tmpFile;
1723          DefaultMutableTreeNode tmpDefNode;
1724
1725          String dirList[] = dir.list();
1726          for (int idx = 0; idx < dirList.length; idx++)
1727          {
1728             tempList.addElement(dirList[idx]);
1729          }
1730
1731          for (Enumeration enum = tempList.elements(); enum.hasMoreElements(); )
1732          {
1733             tmpFile = new File(dir.getPath(), (String)enum.nextElement());
1734             fileList.addElement(tmpFile);
1735
1736             if (tmpFile.isDirectory())
1737             {
1738                DefaultMutableTreeNode dirNode = recurseDir(tmpFile);
1739
1740                if (scanDir(dirNode) == false)
1741                {
1742                   node.add(dirNode);
1743                }
1744             }
1745
1746             if (isPlayListEntry(tmpFile))
1747             {
1748                // If a '-' exists in the filename, then the ArtistName is the first char. to the last char.
1749                // before the '-'.  If the previous Artist of this mp3 does not match
1750                // this one, then increment the ArtistCount variable.
1751
1752                iDashIndex = tmpFile.getName().indexOf('-');
1753                if (iDashIndex > 0)
1754                   currArtist = tmpFile.getName().substring(0, iDashIndex );
1755                else
1756                   currArtist = tmpFile.getName();
1757
1758                if (!currArtist.equalsIgnoreCase(prevArtist))
1759                {
1760                    _iArtistCount++;
1761                    prevArtist = currArtist;
1762                }
1763
1764                tmpDefNode = new DefaultMutableTreeNode(tmpFile.getName());
1765
1766                _iSongCount = _iSongCount + 1;
1767                tmpDefNode.setUserObject(new PlayListEntry(tmpFile.getPath(), _iArtistCount));
1768
1769                tempNode.addElement(tmpDefNode);
1770
1771
1772                // Now, add the new node to the tree in "track" order if possible.  That is, each
1773                // song is of the form "Artist-xx-Song.mp3" where xx is the track number.  (01,02,...,n)
1774                // Otherwise, add the node in alphabetic order.
1775
1776                if (bFirstChildCreated == false)
1777                {
1778                   node.add(tmpDefNode);
1779                   firstChild = new DefaultMutableTreeNode(node.getFirstChild());
1780                   bFirstChildCreated =  true;
1781                }
1782                else
1783                {
1784                   // Start at the first child and compare Artist names, inserting the new child at
1785                   // the top if lowest, before a node if lower than it, or at the end if greater than all
1786                   // void insert(MutableTreeNode newChild, int childIndex)
1787
1788                   boolean bDone = false;
1789                   DefaultMutableTreeNode tmpNode;
1790                   int iTmpIndex;
1791                   int iTrackNum;
1792                   int iTmpTrackNum;
1793                   PlayListEntry tmpObject;
1794                   PlayListEntry newObject = (PlayListEntry)tmpDefNode.getUserObject();
1795
1796                   for (Enumeration e = node.children(); (e.hasMoreElements() && !bDone) ; )
1797                   {
1798                      tmpNode = (DefaultMutableTreeNode)e.nextElement();
```

```
                    tmpObject = (PlayListEntry)tmpNode.getUserObject();

                    // See if there is a track number associated with this song.
                    iTrackNum = newObject.getTrackNum();
                    if (iTrackNum == -1)
                    {
                       // Add by alphabetic order.
                       if (newObject.getArtist().compareTo(tmpObject.getArtist()) == 0)
                       {
                          if (newObject.getArtistIndex() != tmpObject.getArtistIndex())
                          {
                             newObject.setArtistIndex(tmpObject.getArtistIndex());
                          }

                          iTmpIndex = node.getIndex(tmpNode);
                          node.insert(tmpDefNode, iTmpIndex++);
                          bDone = true;
                       }
                       else
                       {
                          if (newObject.getArtist().compareTo(tmpObject.getArtist()) < 0)
                          {
                             //need to bump up all artist index counts {ugh}

                             iTmpIndex = node.getIndex(tmpNode);
                             node.insert(tmpDefNode, iTmpIndex--);
                             bDone = true;
                          }
                       }
                    }
                    else
                    {
                       // Add by "track" order.
                       iTmpTrackNum = tmpObject.getTrackNum();
                       if (iTrackNum < iTmpTrackNum)
                       {
                          iTmpIndex = node.getIndex(tmpNode);
                          node.insert(tmpDefNode, iTmpIndex--);
                          bDone = true;
                       }
                    }
                 }

                 // The current artist is lexically greater than all the other artists.
                 if (bDone == false)
                 {
                    node.add(tmpDefNode);
                 }
              }
           }
        }
        return node;
     }

     /*
      * The following method creates a tree structure corresponding to
      * (initially drive D:) corresponding to a directory structure
      * and returns the tree to the calling method.
      */
     public DefaultTreeModel createTree()
     {
        DefaultMutableTreeNode root = null;

        if (readTreeFromDisk() == true)
        {
           root = (DefaultMutableTreeNode)treeModel.getRoot();
        }
        else
        {
           root = new DefaultMutableTreeNode("Root");
           treeModel = new DefaultTreeModel(root);
        }
        return treeModel;
     }

     public void resetAlreadyPlayedFlags(boolean bWriteToFile)
     {
        DefaultMutableTreeNode root;
        DefaultMutableTreeNode tmpNode;
        PlayListEntry          tmpObject;
        int                    iTotalPlayed = 0;
        int                    iCnt = 0;
        String                 footer1;
        String                 tmpString;

        String header1 = "Paid Count:   Song Name:";
        String header2 = "-------------------------------------------------------------------------";

        root = (DefaultMutableTreeNode)treeModel.getRoot();
```

```
1889        try
1890        {
1891           BufferedWriter out = new BufferedWriter(new FileWriter("MP3Jukebox.REP"));
1892
1893           if (bWriteToFile == true)
1894           {
1895              out.write(header1, 0, header1.length());
1896              out.newLine();
1897              out.write(header2, 0, header2.length());
1898              out.newLine();
1899
1900              iTotalPlayed = 0;
1901           }
1902
1903           // Perform a Prefix traversal of the tree.  For any leaf nodes, reset the played flag to FALSE.
1904           for (Enumeration e = root.preorderEnumeration(); e.hasMoreElements(); )
1905           {
1906              tmpNode = (DefaultMutableTreeNode)e.nextElement();
1907
1908              if (isPlayListEntry(tmpNode))
1909              {
1910                 tmpObject = (PlayListEntry)tmpNode.getUserObject();
1911
1912                 if (bWriteToFile == true)
1913                 {
1914                    iCnt = tmpObject.getPaidCnt();
1915                    iTotalPlayed = iTotalPlayed + iCnt;
1916
1917                    tmpString = new String() + iCnt;
1918
1919                    switch (tmpString.length())
1920                    {
1921                       case 1:
1922                          tmpString = "          " + tmpString + "    ";
1923                          break;
1924
1925                       case 2:
1926                          tmpString = "         " + tmpString + "    ";
1927                          break;
1928
1929                       case 3:
1930                          tmpString = "        " + tmpString + "    ";
1931                          break;
1932
1933                       case 4:
1934                          tmpString = "       " + tmpString + "    ";
1935                          break;
1936
1937                       default:
1938                          tmpString = "      " + tmpString + "    ";
1939                    }
1940
1941                    tmpString = tmpString + tmpObject.toString();
1942
1943                    out.write(tmpString, 0, tmpString.length());
1944                    out.newLine();
1945                 }
1946
1947                 tmpObject.resetPlayedCnt();
1948                 tmpObject.resetPaidCnt();
1949              }
1950           }
1951           if (bWriteToFile == true)
1952           {
1953              out.write(header2, 0, header2.length());
1954              out.newLine();
1955              footer1 = "Total: " + iTotalPlayed;
1956              out.write(footer1, 0, footer1.length());
1957              out.newLine();
1958
1959              out.flush();
1960              out.close();
1961           }
1962        }
1963        catch (java.io.IOException ioException1)
1964        {
1965           try
1966           {
1967              // In case we are running from a CD-ROM, create our directory.
1968              try
1969              {
1970                 File dir = new File("c:/MP3Jukebox");
1971
1972                 if (!dir.exists())
1973                 {
1974                    dir.mkdir();
1975                 }
1976              }
1977              catch (SecurityException excptn)
1978              {
```

```
1979                System.out.println(excptn.toString());
1980             }
1981
1982
1983             BufferedWriter out = new BufferedWriter(new FileWriter("c:/Mp3Jukebox/MP3Jukebox.REP"));
1984
1985             if (bWriteToFile == true)
1986             {
1987                out.write(header1, 0, header1.length());
1988                out.newLine();
1989                out.write(header2, 0, header2.length());
1990                out.newLine();
1991
1992                iTotalPlayed = 0;
1993             }
1994
1995             // Perform a Prefix traversal of the tree.  For any leaf nodes, reset the played flag to FALSE.
1996             for (Enumeration e2 = root.preorderEnumeration(); e2.hasMoreElements(); )
1997             {
1998                tmpNode = (DefaultMutableTreeNode)e2.nextElement();
1999
2000                if (isPlayListEntry(tmpNode))
2001                {
2002                   tmpObject = (PlayListEntry)tmpNode.getUserObject();
2003
2004                   if (bWriteToFile == true)
2005                   {
2006                      iCnt = tmpObject.getPaidCnt();
2007                      iTotalPlayed = iTotalPlayed + iCnt;
2008
2009                      tmpString = new String() + iCnt;
2010
2011                      switch (tmpString.length())
2012                      {
2013                         case 1:
2014                            tmpString = "          " + tmpString + "    ";
2015                            break;
2016
2017                         case 2:
2018                            tmpString = "         " + tmpString + "    ";
2019                            break;
2020
2021                         case 3:
2022                            tmpString = "        " + tmpString + "    ";
2023                            break;
2024
2025                         case 4:
2026                            tmpString = "       " + tmpString + "    ";
2027                            break;
2028
2029                         default:
2030                            tmpString = "      " + tmpString + "    ";
2031                      }
2032
2033                      tmpString = tmpString + tmpObject.toString();
2034
2035                      out.write(tmpString, 0, tmpString.length());
2036                      out.newLine();
2037
2038                      tmpObject.resetPlayedCnt();
2039                      tmpObject.resetPaidCnt();
2040                   }
2041                }
2042             }
2043
2044             if (bWriteToFile == true)
2045             {
2046                out.write(header2, 0, header2.length());
2047                out.newLine();
2048                footer1 = "Total: " + iTotalPlayed;
2049                out.write(footer1, 0, footer1.length());
2050                out.newLine();
2051
2052                out.flush();
2053                out.close();
2054             }
2055          }
2056          catch (java.io.IOException ioException2)
2057          {
2058             System.out.println("ERROR:  Could not write data to disk!");
2059          }
2060       }
2061    }
2062
2063    public void resetAllQueuedCnts()
2064    {
2065       DefaultMutableTreeNode root;
2066       DefaultMutableTreeNode tmpNode;
2067       PlayListEntry          tmpObject;
2068
```

```
2069          _iSongCount = 0;
2070
2071          root = (DefaultMutableTreeNode)treeModel.getRoot();
2072
2073          // Perform a Prefix traversal of the tree.  For any leaf nodes, reset the played flag to FALSE.
2074          for (Enumeration e = root.preorderEnumeration(); e.hasMoreElements(); )
2075          {
2076             tmpNode = (DefaultMutableTreeNode)e.nextElement();
2077
2078             if (isPlayListEntry(tmpNode))
2079             {
2080                tmpObject = (PlayListEntry)tmpNode.getUserObject();
2081                tmpObject.resetQueuedCnt();
2082                _iSongCount = _iSongCount + 1;
2083
2084                //System.out.println(_iSongCount + ": " + tmpNode.toString());
2085             }
2086          }
2087       }
2088
2089       public void resetPaidFlags()
2090       {
2091          DefaultMutableTreeNode root;
2092          DefaultMutableTreeNode tmpNode;
2093          PlayListEntry          tmpObject;
2094
2095          root = (DefaultMutableTreeNode)treeModel.getRoot();
2096
2097          // Perform a Prefix traversal of the tree.  For any leaf nodes, reset the played flag to FALSE.
2098          for (Enumeration e = root.preorderEnumeration(); e.hasMoreElements(); )
2099          {
2100             tmpNode = (DefaultMutableTreeNode)e.nextElement();
2101
2102             if (isPlayListEntry(tmpNode))
2103             {
2104                tmpObject = (PlayListEntry)tmpNode.getUserObject();
2105
2106
2107                // Dump out how many times the song was played...
2108                //System.out.println(tmpObject.toString() + ": " + tmpObject.getPlayedCnt());
2109
2110
2111                tmpObject.resetPaidCnt();
2112             }
2113          }
2114       }
2115
2116       public String getSongCount()
2117       {
2118          Integer intSongCount = new Integer(_iSongCount);
2119
2120          return intSongCount.toString();
2121       }
2122
2123       public int getIntSongCount()
2124       {
2125          return _iSongCount;
2126       }
2127
2128       public int getIntValueSongCount()
2129       {
2130          return _iSongCount;
2131       }
2132
2133       public void writeTreeToDisk()
2134       {
2135           try
2136           {
2137              FileOutputStream f = new FileOutputStream("MP3Jukebox.DAT");
2138              BufferedOutputStream buf = new BufferedOutputStream(f);
2139              ObjectOutputStream  s  =  new  ObjectOutputStream(buf);
2140
2141              s.writeObject(treeModel);
2142              s.flush();
2143              s.close();
2144
2145              f.flush();
2146              f.close();
2147
2148              s = null;
2149              f = null;
2150           }
2151           catch (java.io.IOException e)
2152           {
2153              try
2154              {
2155                 // In case we are running from a CD-ROM, create our directory.
2156                 try
2157                 {
2158                    File dir = new File("c:/MP3Jukebox");
```

```
2159
2160                if (!dir.exists())
2161                {
2162                   dir.mkdir();
2163                }
2164             }
2165             catch (SecurityException excptn)
2166             {
2167                System.out.println(excptn.toString());
2168             }
2169
2170             FileOutputStream f = new FileOutputStream("c:/MP3Jukebox/MP3Jukebox.DAT");
2171             BufferedOutputStream buf = new BufferedOutputStream(f);
2172             ObjectOutputStream  s  =  new  ObjectOutputStream(buf);
2173
2174             s.writeObject(treeModel);
2175             s.flush();
2176             s.close();
2177
2178             f.flush();
2179             f.close();
2180
2181             s = null;
2182             f = null;
2183          }
2184          catch (java.io.IOException e2)
2185          {
2186             System.out.println("ERROR:  Could not write data to disk!");
2187          }
2188       }
2189    }
2190
2191    public boolean readTreeFromDisk()
2192    {
2193       try
2194       {
2195          FileInputStream in = null;
2196          BufferedInputStream buf= null;
2197          ObjectInputStream s = null;
2198
2199          try
2200          {
2201             in = new FileInputStream("MP3Jukebox.DAT");
2202             buf = new BufferedInputStream(in);
2203             s = new ObjectInputStream(buf);
2204          }
2205          catch (java.io.IOException e)
2206          {
2207             in = new FileInputStream("c:/MP3Jukebox/MP3Jukebox.DAT");
2208             buf = new BufferedInputStream(in);
2209             s = new ObjectInputStream(buf);
2210          }
2211
2212          try
2213          {
2214             treeModel = (DefaultTreeModel)s.readObject();
2215
2216             // Verify that the corresponding mp3 files exist.
2217             DefaultMutableTreeNode root = (DefaultMutableTreeNode)treeModel.getRoot();
2218             DefaultMutableTreeNode tmpNode = null;
2219             DefaultMutableTreeNode parent = null;
2220             Vector delVector  = new Vector();
2221             Vector delVector2 = new Vector();
2222
2223             // Perform a traversal of the tree.
2224             for (Enumeration e = root.breadthFirstEnumeration(); e.hasMoreElements(); )
2225             {
2226                tmpNode = (DefaultMutableTreeNode)e.nextElement();
2227
2228                if (!tmpNode.isLeaf() && !tmpNode.isRoot())
2229                {
2230                   if (scanDir(tmpNode))
2231                   {
2232                      delVector.addElement(tmpNode);
2233                   }
2234                }
2235             }
2236
2237             // Remove the empty nodes.
2238             for (Enumeration e = delVector.elements(); e.hasMoreElements(); )
2239             {
2240                tmpNode = (DefaultMutableTreeNode)e.nextElement();
2241                tmpNode.removeFromParent();
2242             }
2243
2244
2245             // Now, prune the tree for any non-mp3 containing nodes/leafs...
2246             for (Enumeration e = root.children(); e.hasMoreElements(); )
2247             {
2248                tmpNode = (DefaultMutableTreeNode)e.nextElement();
```

```
2249
2250                if (!isPlayListEntry(tmpNode.getLastLeaf()))
2251                {
2252                   delVector2.addElement(tmpNode);
2253                }
2254             }
2255
2256             // Remove the empty nodes.
2257             for (Enumeration e = delVector2.elements(); e.hasMoreElements(); )
2258             {
2259                tmpNode = (DefaultMutableTreeNode)e.nextElement();
2260                tmpNode.removeFromParent();
2261             }
2262          }
2263          catch (java.lang.ClassNotFoundException e2)
2264          {
2265             return false;
2266          }
2267          in.close();
2268          s.close();
2269          in = null;
2270          s  = null;
2271
2272       }
2273       catch (java.io.IOException e)
2274       {
2275          return false;
2276       }
2277
2278       return true;
2279    }
2280
2281    public Vector search(String strSearch)
2282    {
2283       DefaultMutableTreeNode root = (DefaultMutableTreeNode)treeModel.getRoot();
2284       DefaultMutableTreeNode node         = null;
2285       Object                 userObject   = null;
2286       Vector                 searchVector = null;
2287
2288       // Perform a Prefix traversal of the tree.
2289       for (Enumeration e = root.preorderEnumeration(); e.hasMoreElements(); )
2290       {
2291          node = (DefaultMutableTreeNode)e.nextElement();
2292          userObject = node.getUserObject();
2293
2294          if (userObject != null && userObject instanceof PlayListEntry)
2295          {
2296             if (((PlayListEntry)userObject).getMp3Path().toLowerCase().indexOf(strSearch.toLowerCase()) >= 0)
2297             {
2298                if (searchVector == null)
2299                   searchVector = new Vector();
2300                searchVector.addElement(userObject);
2301             }
2302          }
2303       }
2304       return searchVector;
2305    }
2306
2307    public Vector search(String strSearch, int iSearchType, int iBooleanOperator, int iMaxSize)
2308    {
2309       DefaultMutableTreeNode root = (DefaultMutableTreeNode)treeModel.getRoot();
2310       DefaultMutableTreeNode node             = null;
2311       Object                 userObject       = null;
2312       Vector                 searchVector     = null;
2313       String                 strTextToCompare = null;
2314       boolean                bDone            = false;
2315       boolean                bHit             = false;
2316
2317       // Perform a Prefix traversal of the tree.
2318       for (Enumeration e = root.preorderEnumeration(); e.hasMoreElements() && !bDone; )
2319       {
2320          bHit = false;
2321          node = (DefaultMutableTreeNode)e.nextElement();
2322          userObject = node.getUserObject();
2323
2324          if (userObject != null && userObject instanceof PlayListEntry)
2325          {
2326             // First, get the test we will be searching in.
2327             switch (iSearchType)
2328             {
2329                case BY_ARTIST:
2330                   strTextToCompare = ((PlayListEntry)userObject).getArtist();
2331                   break;
2332
2333                case BY_CDTITLE:
2334                   strTextToCompare = ((PlayListEntry)userObject).getCDTitle();
2335                   break;
2336
2337                case BY_SONG:
2338                   strTextToCompare = ((PlayListEntry)userObject).getSong();
```

```
                    break;

                default:
                    strTextToCompare = ((PlayListEntry)userObject).getMp3Path();
                    break;
            }



            // TDM:  Need to implement the following...
            // Then, either do an AND or OR search on the passed in tokens.
            if (iBooleanOperator == BOOLEAN_AND)
            {
                if (strTextToCompare.toLowerCase().indexOf(strSearch.toLowerCase()) >= 0)
                    bHit = true;
            }
            else
            {
                if (strTextToCompare.toLowerCase().indexOf(strSearch.toLowerCase()) >= 0)
                    bHit = true;
            }



            // Finally, if there's a hit, add the song to the vector if the vector isn't full.
            if (bHit == true)
            {
                if (searchVector == null)
                {
                    searchVector = new Vector();
                }

                if (searchVector.size() < iMaxSize)
                {
                    searchVector.addElement(userObject);
                }
                else
                {
                    bDone = true;
                }
            }
        }
    }

    return searchVector;
}

private boolean scanDir(DefaultMutableTreeNode parent)
{
    // Verify that the corresponding mp3 files exist.
    DefaultMutableTreeNode node       = null;
    Object                 userObject = null;
    File                   file       = null;
    Vector                 delVector  = new Vector();

    // Perform a Prefix traversal of the tree.
    for (Enumeration e = parent.children(); e.hasMoreElements(); )
    {
        node = (DefaultMutableTreeNode)e.nextElement();
        userObject = node.getUserObject();

        if (userObject != null && userObject instanceof PlayListEntry)
        {
            file = new File(((PlayListEntry)userObject).getMp3Path());
            if (!file.exists())
            {
                delVector.addElement(node);
            }
        }
    }

    // Now, delete the nodes.
    for (Enumeration e = delVector.elements(); e.hasMoreElements(); )
    {
        node = (DefaultMutableTreeNode)e.nextElement();
        parent.remove(node);
    }

    // See if the parent node is empty.
    if (!isPlayListEntry(parent.getLastLeaf()))
        return true;
    else
        return false;
}


private boolean scanNodeForMp3s(DefaultMutableTreeNode parent)
{
    // If at least one mp3 node is found, then return true.
    DefaultMutableTreeNode node       = null;
```

```
2429        Object                  userObject  = null;
2430
2431        // Perform a Prefix traversal of the tree.
2432        boolean bDone = false;
2433        for (Enumeration e = parent.preorderEnumeration(); e.hasMoreElements() && !bDone; )
2434        {
2435           node = (DefaultMutableTreeNode)e.nextElement();
2436           userObject = node.getUserObject();
2437
2438           if (userObject != null && userObject instanceof PlayListEntry)
2439           {
2440              bDone = true;
2441           }
2442        }
2443
2444        return bDone;
2445     }
2446
2447     public boolean isPlayListEntry(File file)
2448     {
2449        boolean bIsPlayListEntry = false;
2450
2451        if (file.isFile())
2452        {
2453           if (_bNumberCDs == true)
2454           {
2455              if (file.toString().endsWith(".mp3"))
2456              {
2457                 bIsPlayListEntry = true;
2458              }
2459           }
2460           else
2461           {
2462              if (file.toString().endsWith(".mp3") || file.toString().endsWith(".wav"))
2463              {
2464                 bIsPlayListEntry = true;
2465              }
2466           }
2467        }
2468
2469        return bIsPlayListEntry;
2470     }
2471
2472     public boolean isPlayListEntry(DefaultMutableTreeNode node)
2473     {
2474        boolean bIsPlayListEntry = false;
2475        Object  userObject       = node.getUserObject();
2476
2477        if (userObject != null && userObject instanceof PlayListEntry)
2478        {
2479           if (((PlayListEntry)userObject).getMp3Path() != null)
2480           {
2481              bIsPlayListEntry = true;
2482           }
2483        }
2484
2485        return bIsPlayListEntry;
2486     }
2487
2488     public void deleteSubTree(DefaultMutableTreeNode node, JTree tree)
2489     {
2490        if (node != null)
2491        {
2492           String strPath = null;
2493
2494           // Get the location of the currently selected node.
2495           int iRow = tree.getRowForPath(new TreePath(node.getPath()));
2496
2497
2498           // Delete the corresponding files/directories on disk.
2499           for (Enumeration e = node.preorderEnumeration(); e.hasMoreElements(); )
2500           {
2501              Object tmpObject;
2502              Object tmpNode = e.nextElement();
2503
2504              if (tmpNode instanceof DefaultMutableTreeNode)
2505              {
2506                 if (isPlayListEntry((DefaultMutableTreeNode)tmpNode))
2507                 {
2508                    tmpObject = ((DefaultMutableTreeNode)tmpNode).getUserObject();
2509
2510                    // Get the path for the cover art .jpg (and possibly .gif files).
2511                    if (strPath == null)
2512                    {
2513                       strPath = ((PlayListEntry)tmpObject).getMp3Path();
2514                    }
2515
2516                    // Now, delete the mp3 from disk.
2517                    try
2518                    {
```

```
                    File file = new File(((PlayListEntry)tmpObject).getMp3Path());

                    file.delete();
                 }
                 catch (SecurityException excptn)
                 {
                    System.out.println(excptn.toString());
                 }
              }
           }
        }


        // Remove the cover art if it exists.
        if (strPath != null)
        {
           String strCover1 = null;
           String strCover2 = null;

           int iSlashIndex = strPath.lastIndexOf("\\");

           if (iSlashIndex != -1)
           {
              strCover1 = strPath.substring(0, iSlashIndex + 1) + "cover.jpg";
              strCover2 = strPath.substring(0, iSlashIndex + 1) + "cover.gif";

              try
              {
                 File file1 = new File(strCover1);
                 if (file1.exists())
                 {
                    file1.delete();
                 }

                 File file2 = new File(strCover2);
                 if (file2.exists())
                 {
                    file2.delete();
                 }

                 file1 = null;
                 file2 = null;
              }
              catch (SecurityException excptn)
              {
                 System.out.println(excptn.toString());
              }
           }
        }


        // Remove the node.
        node.removeAllChildren();
        node.removeFromParent();


        // Refresh the tree.
        reloadTree(tree, iRow);
     }
  }

  public void markAllChildrenPlayed(DefaultMutableTreeNode node)
  {
     DefaultMutableTreeNode tmpNode;

     for (Enumeration enum = node.preorderEnumeration(); enum.hasMoreElements(); )
     {
        tmpNode = (DefaultMutableTreeNode)enum.nextElement();

        if (isPlayListEntry(tmpNode))
        {
           PlayListEntry mp3 = (PlayListEntry)tmpNode.getUserObject();

           mp3.incrementPlayedCnt();
        }
     }
  }

  public void resetAllChildrenPlayed(DefaultMutableTreeNode node)
  {
     DefaultMutableTreeNode tmpNode;

     for (Enumeration enum = node.preorderEnumeration(); enum.hasMoreElements(); )
     {
        tmpNode = (DefaultMutableTreeNode)enum.nextElement();

        if (isPlayListEntry(tmpNode))
        {
           PlayListEntry mp3 = (PlayListEntry)tmpNode.getUserObject();
```

```
2609                mp3.resetPlayedCnt();
2610             }
2611          }
2612       }
2613
2614       public void resetAll(DefaultMutableTreeNode node)
2615       {
2616          DefaultMutableTreeNode tmpNode;
2617
2618          for (Enumeration enum = node.preorderEnumeration(); enum.hasMoreElements(); )
2619          {
2620             tmpNode = (DefaultMutableTreeNode)enum.nextElement();
2621
2622             if (isPlayListEntry(tmpNode))
2623             {
2624                PlayListEntry mp3 = (PlayListEntry)tmpNode.getUserObject();
2625
2626                mp3.resetPlayedCnt();
2627                mp3.resetPaidCnt();
2628             }
2629          }
2630       }
2631
2632       public Vector getAllChildren(DefaultMutableTreeNode node)
2633       {
2634          DefaultMutableTreeNode tmpNode;
2635          PlayListEntry mp3;
2636          Vector vector = new Vector();
2637
2638          for (Enumeration enum = node.preorderEnumeration(); enum.hasMoreElements(); )
2639          {
2640             tmpNode = (DefaultMutableTreeNode)enum.nextElement();
2641
2642             if (isPlayListEntry(tmpNode))
2643             {
2644                mp3 = (PlayListEntry)tmpNode.getUserObject();
2645                vector.addElement(mp3);
2646             }
2647          }
2648          return vector;
2649       }
2650
2651       public Vector getCDAllChildren(DefaultMutableTreeNode node)
2652       {
2653          DefaultMutableTreeNode childNode;
2654
2655          boolean bTruncate = false;
2656          DefaultMutableTreeNode firstChild;
2657          DefaultMutableTreeNode secondChild;
2658          DefaultMutableTreeNode lastChild;
2659
2660          PlayListEntry mp3;
2661          PlayListEntry firstMp3;
2662          PlayListEntry secondMp3;
2663          PlayListEntry lastMp3;
2664
2665          String strSong      = null;
2666          String firstArtist  = null;
2667          String secondArtist = null;
2668          String lastArtist   = null;
2669
2670          Vector vector = new Vector();
2671
2672          try
2673          {
2674             firstChild = (DefaultMutableTreeNode)node.getFirstChild();
2675             if (isPlayListEntry(firstChild))
2676             {
2677                firstMp3 = (PlayListEntry)firstChild.getUserObject();
2678                firstArtist = firstMp3.getArtist();
2679             }
2680
2681
2682             secondChild = (DefaultMutableTreeNode)node.getChildAt(1);
2683             if (isPlayListEntry(secondChild))
2684             {
2685                secondMp3 = (PlayListEntry)secondChild.getUserObject();
2686                secondArtist = secondMp3.getArtist();
2687             }
2688
2689
2690             lastChild  = (DefaultMutableTreeNode)node.getLastChild();
2691             if (isPlayListEntry(lastChild))
2692             {
2693                lastMp3 = (PlayListEntry)lastChild.getUserObject();
2694                lastArtist = lastMp3.getArtist();
2695             }
2696          }
2697          catch (java.lang.ArrayIndexOutOfBoundsException idxExc)
2698          {
```

```
2699
2700        }
2701
2702        if (firstArtist != null && secondArtist != null && lastArtist != null)
2703        {
2704           if (firstArtist.equalsIgnoreCase(lastArtist))
2705           {
2706              if (firstArtist.equalsIgnoreCase(secondArtist))
2707              {
2708                 bTruncate = true;
2709              }
2710           }
2711        }
2712        else
2713        {
2714           // This must be a "Single Song" CD (literally, only one song was found).
2715           if (secondArtist == null)
2716           {
2717              bTruncate = true;
2718           }
2719        }
2720
2721
2722        for (Enumeration enum = node.preorderEnumeration(); enum.hasMoreElements(); )
2723        {
2724           strSong = null;
2725           byte nameIdx = 0;
2726           childNode = (DefaultMutableTreeNode)enum.nextElement();
2727
2728           if (isPlayListEntry(childNode))
2729           {
2730              mp3 = (PlayListEntry)childNode.getUserObject();
2731
2732              if (bTruncate == true)
2733              {
2734                 int iDashIndex = mp3.getTrackDashIndex();
2735                 nameIdx = ((byte)(iDashIndex+1));
2736              }
2737              else
2738              {
2739                 nameIdx = 0;
2740              }
2741
2742              mp3.setNameIdx(nameIdx);
2743              vector.addElement(mp3);
2744           }
2745        }
2746
2747        return vector;
2748     }
2749
2750     public String getGenre(DefaultMutableTreeNode node)
2751     {
2752        DefaultMutableTreeNode parentNode = (DefaultMutableTreeNode)node.getParent();
2753        DefaultMutableTreeNode grandParentNode = null;
2754        String strGenre = null;
2755
2756        if (parentNode.toString().toLowerCase().indexOf("soundtrack") == -1)
2757        {
2758           grandParentNode = (DefaultMutableTreeNode)parentNode.getParent();
2759           strGenre = grandParentNode.toString();
2760        }
2761        else
2762           strGenre = parentNode.toString();
2763
2764        return strGenre;
2765     }
2766
2767     public String getCDArtist(DefaultMutableTreeNode node)
2768     {
2769        DefaultMutableTreeNode parentNode = (DefaultMutableTreeNode)node.getParent();
2770
2771        return parentNode.toString();
2772     }
2773
2774     public String getCDNumberForSong(JTree tree, PlayListEntry mp3)
2775     {
2776        String strCDNum = " ";
2777
2778        DefaultMutableTreeNode root = (DefaultMutableTreeNode)tree.getModel().getRoot();
2779        DefaultMutableTreeNode tmpNode = null;
2780        DefaultMutableTreeNode parentNode = null;
2781
2782        PlayListEntry srchMp3 = null;
2783
2784        boolean bDone  = false;
2785
2786        for (Enumeration enum = root.preorderEnumeration(); enum.hasMoreElements() && !bDone; )
2787        {
2788           tmpNode = (DefaultMutableTreeNode)enum.nextElement();
```

```
2789
2790            if (isPlayListEntry(tmpNode))
2791            {
2792               srchMp3 = (PlayListEntry)tmpNode.getUserObject();
2793
2794               if (mp3.getMp3Path().equalsIgnoreCase(srchMp3.getMp3Path()))
2795               {
2796                  bDone = true;
2797
2798                  parentNode = (DefaultMutableTreeNode)tmpNode.getParent();
2799
2800                  strCDNum = parentNode.toString().substring(0,3);
2801               }
2802            }
2803         }
2804
2805         return strCDNum;
2806      }
2807
2808      public Vector getAllCDNodeChildren(JTree tree)
2809      {
2810         DefaultMutableTreeNode root = (DefaultMutableTreeNode)tree.getModel().getRoot();
2811
2812         DefaultMutableTreeNode tmpNode;
2813         DefaultMutableTreeNode childNode;
2814
2815         int     iDashIndex;
2816         String  currentCD;
2817         Integer intRow;
2818         int     iRow;
2819
2820         Vector vector = new Vector();
2821
2822         for (Enumeration enum = root.preorderEnumeration(); enum.hasMoreElements(); )
2823         {
2824            tmpNode = (DefaultMutableTreeNode)enum.nextElement();
2825
2826            // e.g. For "001-The Greatest Hits", a dash would be at position 3.
2827            iDashIndex = tmpNode.toString().indexOf('-');
2828            if ((iDashIndex == 3) && (!tmpNode.isLeaf()))
2829            {
2830               currentCD = tmpNode.toString().substring(0, iDashIndex);
2831
2832               // Get the first child of this node.  If it is an MP3 and a "-" exists in the
2833               // third position of the title, then boldly assume we are at a CDNode, if so,
2834               // add the corresponding JTree row to the return vector.
2835
2836               try
2837               {
2838                  childNode = (DefaultMutableTreeNode)tree.getModel().getChild(tmpNode, 0);
2839                  if (isPlayListEntry(childNode))
2840                  {
2841                     intRow = null;
2842
2843                     iRow = tree.getRowForPath(new TreePath(tmpNode.getPath()));
2844                     intRow = new Integer(iRow);
2845                     vector.addElement(intRow);
2846
2847                     // System.out.println("Adding index: " + iRow + " for: " + tmpNode.toString());
2848                  }
2849               }
2850               catch (java.lang.ArrayIndexOutOfBoundsException idxExc)
2851               {
2852                  System.out.println("No children for: " + tmpNode.toString());
2853               }
2854            }
2855         }
2856         return vector;
2857      }
2858
2859      public Vector getAllGenreNodeChildren(JTree tree)
2860      {
2861         DefaultMutableTreeNode root = (DefaultMutableTreeNode)tree.getModel().getRoot();
2862
2863         DefaultMutableTreeNode tmpNode;
2864         DefaultMutableTreeNode tmpNode2;
2865
2866         Integer intRow;
2867         int     iRow;
2868
2869         Vector vector = new Vector();
2870
2871         for (Enumeration enum = root.children(); enum.hasMoreElements(); )
2872         {
2873            tmpNode = (DefaultMutableTreeNode)enum.nextElement();
2874
2875            for (Enumeration enum2 = tmpNode.children(); enum2.hasMoreElements(); )
2876            {
2877               tmpNode2 = (DefaultMutableTreeNode)enum2.nextElement();
2878
```

```
2879              if (tmpNode2.isLeaf() == false)
2880              {
2881                 intRow = null;
2882
2883                 iRow = tree.getRowForPath(new TreePath(tmpNode2.getPath()));
2884                 intRow = new Integer(iRow);
2885                 vector.addElement(intRow);
2886
2887                 // System.out.println("Adding index: " + iRow + " for: " + tmpNode2.toString());
2888              }
2889           }
2890        }
2891        return vector;
2892     }
2893
2894     public ImageIcon getCoverImage(JTree tree, DefaultMutableTreeNode node)
2895     {
2896        ImageIcon image;
2897        DefaultMutableTreeNode childNode;
2898        Object  userObject;
2899        String  strTemp1;
2900        String  strTemp2;
2901        String  strCover = null;
2902        File    fCover;
2903        boolean bCoverExists = false;
2904
2905        childNode = (DefaultMutableTreeNode)tree.getModel().getChild(node, 0);
2906        if (isPlayListEntry(childNode))
2907        {
2908           userObject = childNode.getUserObject();
2909
2910           strTemp1 = ((PlayListEntry)userObject).getMp3Path();
2911
2912           // int iSlashIndex = strTemp1.lastIndexOf(File.pathSeparatorChar);
2913           int iSlashIndex = strTemp1.lastIndexOf("\\");
2914
2915           if (iSlashIndex != -1)
2916           {
2917              strTemp2 = strTemp1.substring(0, iSlashIndex+1);
2918              strCover = strTemp2 + "cover.jpg";
2919
2920              // System.out.println("strCover: " + strCover);
2921           }
2922        }
2923
2924        if (strCover != null)
2925        {
2926           fCover = new File(strCover);
2927           if (fCover.exists())
2928           {
2929              bCoverExists = true;
2930           }
2931        }
2932
2933        if (bCoverExists)
2934        {
2935           image = new ImageIcon(strCover);
2936        }
2937        else
2938        {
2939           image = new ImageIcon("images/blankcover.gif");
2940        }
2941
2942        return image;
2943     }
2944
2945     public PlayListEntry getSongForJukeboxNo(JTree tree, String strSelection)
2946     {
2947        PlayListEntry mp3 = null;
2948
2949        DefaultMutableTreeNode root = (DefaultMutableTreeNode)tree.getModel().getRoot();
2950
2951        DefaultMutableTreeNode tmpNode;
2952        DefaultMutableTreeNode childNode;
2953
2954        String  strCD = null;
2955        String  strSong = null;
2956        int     iCDIndex;
2957        int     iSongIndex;
2958        boolean bDone = false;
2959        boolean bDone2 = false;
2960
2961
2962        // Determine what we are looking for.
2963        if (strSelection.length() == 5)
2964        {
2965           strCD   = strSelection.substring(0,3) + "-";
2966           strSong = "-" + strSelection.substring(3,5) + "-";
2967
2968           //System.out.println("CD:   " + strCD);
```

```
2969            //System.out.println("Song: " + strSong);
2970         }
2971
2972
2973         // Now, look for it.
2974         for (Enumeration enum = root.preorderEnumeration(); enum.hasMoreElements() && !bDone; )
2975         {
2976            tmpNode = (DefaultMutableTreeNode)enum.nextElement();
2977
2978            iCDIndex = tmpNode.toString().indexOf(strCD);
2979            if (iCDIndex >= 0)
2980            {
2981               bDone = true;
2982
2983               // We found the CD, now search for the track.
2984               for (Enumeration enum2 = tmpNode.children(); enum2.hasMoreElements() && !bDone2; )
2985               {
2986                  childNode = (DefaultMutableTreeNode)enum2.nextElement();
2987
2988                  iSongIndex = childNode.toString().indexOf(strSong);
2989                  if (iSongIndex >= 0)
2990                  {
2991                     bDone2 = true;
2992
2993                     // We found the song
2994                     if (isPlayListEntry(childNode))
2995                     {
2996                        mp3 = (PlayListEntry)childNode.getUserObject();
2997                     }
2998                  }
2999               }
3000            }
3001         }
3002
3003         return mp3;
3004      }
3005
3006      /** Create a vector that is sorted by the number of times that a song has been played.  Only include
3007       *  those songs that have been played at least ONCE...
3008       */
3009      public Vector getRankingVector(JTree tree)
3010      {
3011         Vector vect = new Vector();
3012         DefaultMutableTreeNode root = (DefaultMutableTreeNode)tree.getModel().getRoot();
3013         DefaultMutableTreeNode tmpNode = null;
3014         PlayListEntry mp3     = null;
3015         PlayListEntry srchMp3 = null;
3016         boolean bDone  = false;
3017         boolean bFound = false;
3018         int iCurrPaidCnt = 0;
3019         int iSrchPaidCnt = 0;
3020         int iMaxPaidCnt  = 1;
3021         int iAddIndex    = 0;
3022
3023
3024         for (Enumeration enum = root.preorderEnumeration(); enum.hasMoreElements() && !bDone; )
3025         {
3026            tmpNode = (DefaultMutableTreeNode)enum.nextElement();
3027
3028            if (isPlayListEntry(tmpNode))
3029            {
3030               mp3 = (PlayListEntry)tmpNode.getUserObject();
3031               iCurrPaidCnt = mp3.getPaidCnt();
3032
3033               if (iCurrPaidCnt > 0)
3034               {
3035                  if (iCurrPaidCnt > iMaxPaidCnt)
3036                  {
3037                     // This is the most played song that has been found yet.
3038                     iMaxPaidCnt = iCurrPaidCnt;
3039                     vect.insertElementAt(mp3, 0);
3040                     //System.out.println(iCurrPaidCnt + ": Adding to beginning: " + mp3.toString());
3041                  }
3042                  else
3043                  {
3044                     // Find out where to insert the song.
3045                     bFound = false;
3046                     for (Enumeration enum2 = vect.elements(); enum2.hasMoreElements() && !bFound; )
3047                     {
3048                        srchMp3 = (PlayListEntry)enum2.nextElement();
3049                        iSrchPaidCnt = srchMp3.getPaidCnt();
3050
3051                        if (iCurrPaidCnt > iSrchPaidCnt)
3052                        {
3053                           bFound = true;
3054                           iAddIndex = vect.lastIndexOf(srchMp3);
3055
3056                           vect.insertElementAt(mp3, iAddIndex);
3057                           //System.out.println(iCurrPaidCnt + ": Inserting at index: " + iAddIndex + ": " + mp3.
toString());
```

```
                        }
                    }

                    if (bFound == false)
                    {
                        vect.addElement(mp3);
                        //System.out.println(iCurrPaidCnt + ": Adding to end: " + mp3.toString());
                    }
                }
            }
        }
    }

    return vect;
}


/** Create a vector that is sorted by the ratio of plays per day of
 *  existence in the treeModel.
 */
public Vector getPowerRankingVector(JTree tree)
{
    Vector vect = new Vector();

    DefaultMutableTreeNode root = (DefaultMutableTreeNode)tree.getModel().getRoot();
    DefaultMutableTreeNode tmpNode = null;

    PlayListEntry mp3     = null;
    PlayListEntry srchMp3 = null;

    boolean bDone  = false;
    boolean bFound = false;

    int iCurrPaidCnt = 0;
    int iSrchPaidCnt = 0;

    int iCurrAge = 0;
    int iSrchAge = 0;

    double dblCurrRatio = 0;
    double dblSrchRatio = 0;
    double dblMaxRatio = 0;

    int iMaxAge     = 0;
    int iMaxPaidCnt = 1;

    int iAddIndex    = 0;


    for (Enumeration enum = root.preorderEnumeration(); enum.hasMoreElements() && !bDone; )
    {
        tmpNode = (DefaultMutableTreeNode)enum.nextElement();

        if (isPlayListEntry(tmpNode))
        {
            mp3 = (PlayListEntry)tmpNode.getUserObject();

            iCurrPaidCnt = mp3.getPaidCnt();
            iCurrAge     = mp3.getAge();

            if (iCurrPaidCnt > 0)
            {

                if (iCurrAge == 0)
                    iCurrAge = 1;

                dblCurrRatio = ((double)iCurrPaidCnt) / ((double)iCurrAge);

                if (dblCurrRatio > dblMaxRatio)
                {
                    // This is the most played song (per day) that has been found yet.
                    dblMaxRatio = dblCurrRatio;

                    vect.insertElementAt(mp3, 0);
                }
                else
                {
                    // Find out where to insert the song.
                    bFound = false;
                    for (Enumeration enum2 = vect.elements(); enum2.hasMoreElements() && !bFound; )
                    {
                        srchMp3 = (PlayListEntry)enum2.nextElement();
                        iSrchPaidCnt = srchMp3.getPaidCnt();
                        iSrchAge     = srchMp3.getAge();

                        if (iSrchAge == 0)
                            iSrchAge = 1;

                        dblSrchRatio = ((double)iSrchPaidCnt) / ((double)iSrchAge);

```

```
3148                     if (dblCurrRatio > dblSrchRatio)
3149                     {
3150                        bFound = true;
3151                        iAddIndex = vect.lastIndexOf(srchMp3);
3152
3153                        vect.insertElementAt(mp3, iAddIndex);
3154                     }
3155                  }
3156
3157                  if (bFound == false)
3158                  {
3159                     vect.addElement(mp3);
3160                  }
3161               }
3162            }
3163         }
3164      }
3165
3166      return vect;
3167   }
3168
3169
3170   /** Create a vector that is sorted by the
3171    *  those songs that have been played at least ONCE...
3172    */
3173   public Vector getCDRankingVector(JTree tree)
3174   {
3175      Vector vect = new Vector();
3176      Vector sumVect = new Vector();
3177
3178      DefaultMutableTreeNode root = (DefaultMutableTreeNode)tree.getModel().getRoot();
3179
3180      DefaultMutableTreeNode tmpNode;
3181      DefaultMutableTreeNode childNode;
3182      DefaultMutableTreeNode srchNode;
3183
3184      PlayListEntry mp3 = null;
3185      boolean bFound = false;
3186
3187      int iCurrSum = 0;
3188      int iSrchSum = 0;
3189      int iMaxSum  = 0;
3190
3191      Integer intCurrSum = new Integer(0);
3192      Integer intSrchSum = new Integer(0);
3193      Integer intMaxSum = new Integer(0);
3194
3195      int     iDashIndex;
3196      int     iAddIndex;
3197      String  currentCD;
3198      Integer intRow;
3199      int     iRow;
3200
3201      for (Enumeration enum = root.preorderEnumeration(); enum.hasMoreElements(); )
3202      {
3203         tmpNode = (DefaultMutableTreeNode)enum.nextElement();
3204
3205         // e.g. For "001-The Greatest Hits", a dash would be at position 3.
3206         iDashIndex = tmpNode.toString().indexOf('-');
3207         if ((iDashIndex == 3) && (!tmpNode.isLeaf()))
3208         {
3209            // Enumerate all the children of this node.  If they are
3210            // MP3's, then get the paid played count for each.  Sort the
3211            // vector by those CD nodes whose sum paid played count is
3212            // greatest (in other words, the most popular CDs).
3213
3214            iCurrSum = 0;
3215            try
3216            {
3217               for (Enumeration enumCD = tmpNode.children(); enumCD.hasMoreElements(); )
3218               {
3219                  childNode = (DefaultMutableTreeNode)enumCD.nextElement();
3220
3221                  if (isPlayListEntry(childNode))
3222                  {
3223                     mp3 = (PlayListEntry)childNode.getUserObject();
3224                     iCurrSum = iCurrSum + mp3.getPaidCnt();
3225                  }
3226               }
3227            }
3228            catch (java.lang.ArrayIndexOutOfBoundsException idxExc)
3229            {
3230               System.out.println("No children for: " + tmpNode.toString());
3231            }
3232
3233            intCurrSum = new Integer(iCurrSum);
3234
3235            // Now, see where this CD fits...
3236            if (iCurrSum > 0)
3237            {
```

```
3238                if (iCurrSum > iMaxSum)
3239                {
3240                   // This is the most played CD that has been found yet.
3241                   iMaxSum = iCurrSum;
3242                   intMaxSum = new Integer(iCurrSum);
3243
3244                   vect.insertElementAt(tmpNode, 0);
3245                   sumVect.insertElementAt(intMaxSum, 0);
3246                }
3247                else
3248                {
3249                   // Find out where to insert the CD.
3250                   bFound = false;
3251                   for (Enumeration enum2 = vect.elements(); enum2.hasMoreElements() && !bFound; )
3252                   {
3253                      srchNode = (DefaultMutableTreeNode)enum2.nextElement();
3254
3255                      iAddIndex = vect.lastIndexOf(srchNode);
3256                      intSrchSum = (Integer)sumVect.elementAt(iAddIndex);
3257                      iSrchSum = intSrchSum.intValue();
3258
3259                      if (iCurrSum > iSrchSum)
3260                      {
3261                         bFound = true;
3262
3263                         vect.insertElementAt(tmpNode, iAddIndex);
3264                         sumVect.insertElementAt(intCurrSum, iAddIndex);
3265                      }
3266                   }
3267
3268                   // Insert at the end.
3269                   if (bFound == false)
3270                   {
3271                      vect.addElement(tmpNode);
3272                      sumVect.addElement(intCurrSum);
3273                   }
3274                }
3275             }
3276          }
3277       }
3278
3279       return vect;
3280    }
3281
3282
3283    /** Create a vector that is sorted by the
3284     *  those songs that have been played at least ONCE...
3285     */
3286    public Vector getCDPowerRankingVector(JTree tree)
3287    {
3288       Vector vect = new Vector();
3289       Vector ratioVect = new Vector();
3290
3291       DefaultMutableTreeNode root = (DefaultMutableTreeNode)tree.getModel().getRoot();
3292
3293       DefaultMutableTreeNode tmpNode;
3294       DefaultMutableTreeNode childNode;
3295       DefaultMutableTreeNode srchNode;
3296
3297       PlayListEntry mp3 = null;
3298       boolean bFound = false;
3299
3300       double dblCurrRatio = 0;
3301       double dblSrchRatio = 0;
3302       double dblMaxRatio = 0;
3303
3304       int iCurrSum = 0;
3305       int iAge = 0;
3306
3307       Double doubleCurrRatio = new Double(0);
3308       Double doubleSrchRatio = new Double(0);
3309       Double doubleMaxRatio = new Double(0);
3310
3311       int     iDashIndex;
3312       int     iAddIndex;
3313       String  currentCD;
3314       Integer intRow;
3315       int     iRow;
3316
3317       for (Enumeration enum = root.preorderEnumeration(); enum.hasMoreElements(); )
3318       {
3319          tmpNode = (DefaultMutableTreeNode)enum.nextElement();
3320
3321          // e.g. For "001-The Greatest Hits", a dash would be at position 3.
3322          iDashIndex = tmpNode.toString().indexOf('-');
3323          if ((iDashIndex == 3) && (!tmpNode.isLeaf()))
3324          {
3325             // Enumerate all the children of this node.  If they are
3326             // MP3's, then get the paid played count for each.  Sort the
3327             // vector by those CD nodes whose sum paid played count is
```

```
3328            // greatest (in other words, the most popular CDs).
3329
3330            iCurrSum = 0;
3331            try
3332            {
3333               for (Enumeration enumCD = tmpNode.children(); enumCD.hasMoreElements(); )
3334               {
3335                  childNode = (DefaultMutableTreeNode)enumCD.nextElement();
3336
3337                  if (isPlayListEntry(childNode))
3338                  {
3339                     mp3 = (PlayListEntry)childNode.getUserObject();
3340                     iCurrSum = iCurrSum + mp3.getPaidCnt();
3341                     iAge = mp3.getAge();
3342                  }
3343               }
3344            }
3345            catch (java.lang.ArrayIndexOutOfBoundsException idxExc)
3346            {
3347               System.out.println("No children for: " + tmpNode.toString());
3348            }
3349
3350            if (iAge == 0)
3351               iAge = 1;
3352
3353            dblCurrRatio = ((double)iCurrSum) / ((double)iAge);
3354
3355            doubleCurrRatio = new Double(dblCurrRatio);
3356
3357            // Now, see where this CD fits...
3358            if (iCurrSum > 0)
3359            {
3360               if (dblCurrRatio > dblMaxRatio)
3361               {
3362                  // This is the most played CD that has been found yet.
3363                  dblMaxRatio = dblCurrRatio;
3364                  doubleMaxRatio = new Double(dblCurrRatio);
3365
3366                  vect.insertElementAt(tmpNode, 0);
3367                  ratioVect.insertElementAt(doubleMaxRatio, 0);
3368               }
3369               else
3370               {
3371                  // Find out where to insert the CD.
3372                  bFound = false;
3373                  for (Enumeration enum2 = vect.elements(); enum2.hasMoreElements() && !bFound; )
3374                  {
3375                     srchNode = (DefaultMutableTreeNode)enum2.nextElement();
3376
3377                     iAddIndex = vect.lastIndexOf(srchNode);
3378                     doubleSrchRatio = (Double)ratioVect.elementAt(iAddIndex);
3379                     dblSrchRatio = doubleSrchRatio.doubleValue();
3380
3381                     if (dblCurrRatio > dblSrchRatio)
3382                     {
3383                        bFound = true;
3384
3385                        vect.insertElementAt(tmpNode, iAddIndex);
3386                        ratioVect.insertElementAt(doubleCurrRatio, iAddIndex);
3387                     }
3388                  }
3389
3390                  // Insert at the end.
3391                  if (bFound == false)
3392                  {
3393                     vect.addElement(tmpNode);
3394                     ratioVect.addElement(doubleCurrRatio);
3395                  }
3396               }
3397            }
3398         }
3399      }
3400
3401      return vect;
3402   }
3403
3404   /** Create a vector that contains all the CDs whose age
3405    *  is less than or equal to iAgeThreshold.
3406    *  This vector is sorted by age (first CD is the youngest).
3407    */
3408   public Vector getNewCDRankingVector(JTree tree, int iNewCDAgeThreshold)
3409   {
3410      Vector vect = new Vector();
3411      Vector ageVect = new Vector();
3412
3413      DefaultMutableTreeNode root = (DefaultMutableTreeNode)tree.getModel().getRoot();
3414
3415      DefaultMutableTreeNode tmpNode;
3416      DefaultMutableTreeNode childNode;
3417      DefaultMutableTreeNode srchNode;
```

```
3419          PlayListEntry mp3 = null;
3420          boolean bFound = false;
3421
3422          int iAge      = 0;
3423          int iSrchAge  = 0;
3424          int iYoungest = 32767;
3425
3426          Integer intAge = new Integer(0);
3427          Integer intSrchAge = new Integer(0);
3428          Integer intYoungest = new Integer(32767);
3429
3430          int     iDashIndex;
3431          int     iAddIndex;
3432          String  currentCD;
3433          Integer intRow;
3434          int     iRow;
3435
3436
3437          for (Enumeration enum = root.preorderEnumeration(); enum.hasMoreElements(); )
3438          {
3439             tmpNode = (DefaultMutableTreeNode)enum.nextElement();
3440
3441             // e.g. For "001-The Greatest Hits", a dash would be at position 3.
3442             iDashIndex = tmpNode.toString().indexOf('-');
3443             if ((iDashIndex == 3) && (!tmpNode.isLeaf()))
3444             {
3445                // Enumerate all the children of this node.  If they are
3446                // MP3's, then get the paid played count for each.  Sort the
3447                // vector by those CD nodes whose sum paid played count is
3448                // greatest (in other words, the most popular CDs).
3449
3450                iAge = 0;
3451                boolean bDone = false;
3452                try
3453                {
3454                   for (Enumeration enumCD = tmpNode.children(); enumCD.hasMoreElements() && bDone == false; )
3455                   {
3456                      childNode = (DefaultMutableTreeNode)enumCD.nextElement();
3457
3458                      if (isPlayListEntry(childNode))
3459                      {
3460                         mp3 = (PlayListEntry)childNode.getUserObject();
3461                         iAge = mp3.getAge();
3462                         intAge = new Integer(iAge);
3463                         bDone = true;
3464                      }
3465                   }
3466                }
3467                catch (java.lang.ArrayIndexOutOfBoundsException idxExc)
3468                {
3469                   System.out.println("No children for: " + tmpNode.toString());
3470                }
3471
3472                // If the CD is young enough, see where it fits.
3473                if (iAge <= iNewCDAgeThreshold)
3474                {
3475                   if (iAge < iYoungest)
3476                   {
3477                      // This is the youngest CD that has been found yet.
3478                      iYoungest = iAge;
3479                      intYoungest = new Integer(iYoungest);
3480
3481                      vect.insertElementAt(tmpNode, 0);
3482                      ageVect.insertElementAt(intYoungest, 0);
3483                   }
3484                   else
3485                   {
3486                      // Find out where to insert the CD.
3487                      bFound = false;
3488                      for (Enumeration enum2 = vect.elements(); enum2.hasMoreElements() && !bFound; )
3489                      {
3490                         srchNode = (DefaultMutableTreeNode)enum2.nextElement();
3491
3492                         iAddIndex = vect.lastIndexOf(srchNode);
3493                         intSrchAge = (Integer)ageVect.elementAt(iAddIndex);
3494                         iSrchAge = intSrchAge.intValue();
3495
3496                         if (iAge < iSrchAge)
3497                         {
3498                            bFound = true;
3499
3500                            vect.insertElementAt(tmpNode, iAddIndex);
3501                            ageVect.insertElementAt(intAge, iAddIndex);
3502                         }
3503                      }
3504
3505                      // Insert at the end.
3506                      if (bFound == false)
3507                      {
3508                         vect.addElement(tmpNode);
```

```
3509                        ageVect.addElement(intAge);
3510                     }
3511                  }
3512               }
3513            }
3514         }
3515
3516         return vect;
3517      }
3518
3519      /** Create a vector that contains all the CDs whose age
3520       *  is less than or equal to iAgeThreshold.
3521       *  This vector is sorted by the the most plays per day.
3522       */
3523      public Vector getNewCDPowerRankingVector(JTree tree, int iAgeThreshold)
3524      {
3525         Vector vect = new Vector();
3526         Vector ratioVect = new Vector();
3527
3528         DefaultMutableTreeNode root = (DefaultMutableTreeNode)tree.getModel().getRoot();
3529
3530         DefaultMutableTreeNode tmpNode;
3531         DefaultMutableTreeNode childNode;
3532         DefaultMutableTreeNode srchNode;
3533
3534         PlayListEntry mp3 = null;
3535         boolean bFound = false;
3536
3537         double dblCurrRatio = 0;
3538         double dblSrchRatio = 0;
3539         double dblMaxRatio = 0;
3540
3541         int iCurrSum = 0;
3542         int iAge = 0;
3543
3544         Double doubleCurrRatio = new Double(0);
3545         Double doubleSrchRatio = new Double(0);
3546         Double doubleMaxRatio = new Double(0);
3547
3548         int     iDashIndex;
3549         int     iAddIndex;
3550         String  currentCD;
3551         Integer intRow;
3552         int     iRow;
3553
3554         for (Enumeration enum = root.preorderEnumeration(); enum.hasMoreElements(); )
3555         {
3556            tmpNode = (DefaultMutableTreeNode)enum.nextElement();
3557
3558            // e.g. For "001-The Greatest Hits", a dash would be at position 3.
3559            iDashIndex = tmpNode.toString().indexOf('-');
3560            if ((iDashIndex == 3) && (!tmpNode.isLeaf()))
3561            {
3562               // Enumerate all the children of this node.  If they are
3563               // MP3's, then get the paid played count for each.  Sort the
3564               // vector by those CD nodes whose sum paid played count is
3565               // greatest (in other words, the most popular CDs).
3566
3567               iCurrSum = 0;
3568               try
3569               {
3570                  for (Enumeration enumCD = tmpNode.children(); enumCD.hasMoreElements(); )
3571                  {
3572                     childNode = (DefaultMutableTreeNode)enumCD.nextElement();
3573
3574                     if (isPlayListEntry(childNode))
3575                     {
3576                        mp3 = (PlayListEntry)childNode.getUserObject();
3577                        iCurrSum = iCurrSum + mp3.getPaidCnt();
3578                        iAge = mp3.getAge();
3579                     }
3580                  }
3581               }
3582               catch (java.lang.ArrayIndexOutOfBoundsException idxExc)
3583               {
3584                  System.out.println("No children for: " + tmpNode.toString());
3585               }
3586
3587               if (iAge == 0)
3588                  iAge = 1;
3589
3590               dblCurrRatio = ((double)iCurrSum) / ((double)iAge);
3591
3592               doubleCurrRatio = new Double(dblCurrRatio);
3593
3594               // If this CD qualifies for age, see where it fits...
3595               if (iAge <= iAgeThreshold)
3596               {
3597                  if (dblCurrRatio > dblMaxRatio)
3598                  {
```

```
3599                    // This is the most played CD that has been found yet.
3600                    dblMaxRatio = dblCurrRatio;
3601                    doubleMaxRatio = new Double(dblCurrRatio);
3602
3603                    vect.insertElementAt(tmpNode, 0);
3604                    ratioVect.insertElementAt(doubleMaxRatio, 0);
3605                 }
3606                 else
3607                 {
3608                    // Find out where to insert the CD.
3609                    bFound = false;
3610                    for (Enumeration enum2 = vect.elements(); enum2.hasMoreElements() && !bFound; )
3611                    {
3612                       srchNode = (DefaultMutableTreeNode)enum2.nextElement();
3613
3614                       iAddIndex = vect.lastIndexOf(srchNode);
3615                       doubleSrchRatio = (Double)ratioVect.elementAt(iAddIndex);
3616                       dblSrchRatio = doubleSrchRatio.doubleValue();
3617
3618                       if (dblCurrRatio > dblSrchRatio)
3619                       {
3620                          bFound = true;
3621
3622                          vect.insertElementAt(tmpNode, iAddIndex);
3623                          ratioVect.insertElementAt(doubleCurrRatio, iAddIndex);
3624                       }
3625                    }
3626
3627                    // Insert at the end.
3628                    if (bFound == false)
3629                    {
3630                       vect.addElement(tmpNode);
3631                       ratioVect.addElement(doubleCurrRatio);
3632                    }
3633                 }
3634              }
3635           }
3636        }
3637
3638        return vect;
3639     }
3640
3641     /** Output a line for each song, where the vital stats are written out, each field separated by a space.
3642      *  The MP3 Name itself will be enclosed within quotes, so that the StringTokenizer will work.
3643      */
3644     public void dumpCDStats()
3645     {
3646        DefaultMutableTreeNode root = (DefaultMutableTreeNode)treeModel.getRoot();;
3647
3648        DefaultMutableTreeNode tmpNode   = null;
3649        PlayListEntry          tmpObject = null;
3650
3651        BufferedWriter out       = null;
3652        String         strLine   = null;
3653
3654        Integer intPaidPlayedCnt = null;
3655        Integer intAgeInDays     = null;     // Since 1/1/2000.
3656        Integer intFreePlayedCnt = null;
3657        String  mp3PathName      = null;
3658
3659        try
3660        {
3661           out = new BufferedWriter(new FileWriter("CDStats.TXT"));
3662
3663           for (Enumeration e = root.preorderEnumeration(); e.hasMoreElements(); )
3664           {
3665              tmpNode = (DefaultMutableTreeNode)e.nextElement();
3666
3667              if (isPlayListEntry(tmpNode))
3668              {
3669                 tmpObject = (PlayListEntry)tmpNode.getUserObject();
3670
3671                 mp3PathName      = tmpObject.getMp3Path();
3672                 intPaidPlayedCnt = new Integer(tmpObject.getPaidCnt());
3673                 intAgeInDays     = new Integer(tmpObject.getRawAge());
3674                 intFreePlayedCnt = new Integer(tmpObject.getPlayedCnt());
3675
3676                 String strPaidPlayed = padString(intPaidPlayedCnt.toString());
3677                 String strAgeInDays  = padString(intAgeInDays.toString());
3678                 String strFreePlayed = padString(intFreePlayedCnt.toString());
3679
3680                 strLine = strPaidPlayed + strAgeInDays + strFreePlayed + "      " + mp3PathName;
3681
3682                 out.write(strLine, 0, strLine.length());
3683                 out.newLine();
3684              }
3685           }
3686
3687           out.flush();
3688           out.close();
```

```
3689         }
3690         catch (java.io.IOException ioException)
3691         {
3692            System.out.println("ERROR:  Could not write data to disk!");
3693         }
3694      }
3695
3696      /**
3697       *
3698       *
3699       */
3700      private String padString(String str)
3701      {
3702         if (str.length() < 6)
3703         {
3704            switch (str.length())
3705            {
3706               case 1:
3707                  str = "     " + str;
3708                  break;
3709
3710               case 2:
3711                  str = "    " + str;
3712                  break;
3713
3714               case 3:
3715                  str = "   " + str;
3716                  break;
3717
3718               case 4:
3719                  str = "  " + str;
3720                  break;
3721
3722               case 5:
3723                  str = " " + str;
3724                  break;
3725
3726               default:
3727            }
3728         }
3729         return str;
3730      }
3731
3732      /**
3733       *
3734       */
3735      private void restoreCDStats()
3736      {
3737         PlayListEntry mp3 = null;
3738
3739         BufferedReader  in        = null;
3740         String          strLine   = null;
3741         StringTokenizer tokenizer = null;
3742
3743         Integer intPaidPlayedCnt = null;
3744         Integer intAgeInDays     = null;
3745         Integer intFreePlayedCnt = null;
3746         String  mp3PathName      = null;
3747
3748         int iColonIndex = -1; // If we ever move to Linux, this needs to be changed.
3749
3750         try
3751         {
3752            File statsFile = new File("CDStats.TXT");
3753
3754            if (statsFile.exists())
3755            {
3756               System.out.println("Attempting to restore statistics...");
3757
3758               in = new BufferedReader(new FileReader(statsFile));
3759
3760               strLine = "";
3761               while (strLine != null)
3762               {
3763                  strLine = in.readLine();
3764
3765                  if (strLine != null)
3766                  {
3767                     iColonIndex = strLine.indexOf(':') - 1; //       +
3768                                                             // e.g. E:\Rock\U2\Achtung Baby\The Fly.mp3
3769
3770                     intPaidPlayedCnt = null;
3771                     intAgeInDays     = null;
3772                     intFreePlayedCnt = null;
3773
3774                     tokenizer = new StringTokenizer(strLine);
3775
3776
3777                     // Get the paid played count.
3778                     intPaidPlayedCnt = new Integer(tokenizer.nextToken());
```

```
3779
3780
3781                    // Get the age.
3782                    intAgeInDays = new Integer(tokenizer.nextToken());
3783
3784
3785                    // Get the free played count
3786                    intFreePlayedCnt = new Integer(tokenizer.nextToken());
3787
3788
3789                    // Get the MP3 pathname.
3790                    mp3PathName = strLine.substring(iColonIndex, strLine.length());
3791
3792
3793                    // Look for the PlayListEntry object.
3794                    mp3 = getPlayListObjForSong(mp3PathName);
3795
3796                    // If we found it (which we should!), then update its vital stats.
3797                    if (mp3 != null)
3798                    {
3799                       mp3.setPlayedCnt(intFreePlayedCnt.intValue());
3800                       mp3.setPaidCnt(intPaidPlayedCnt.intValue());
3801                       mp3.setRawAge(intAgeInDays.intValue());
3802
3803                       if (intPaidPlayedCnt.intValue() > 0)
3804                       {
3805                          System.out.println(" ");
3806                          System.out.println("Song: " + mp3PathName);
3807                          System.out.println("Paid (data): " + intPaidPlayedCnt.toString() + "  (object):" + mp3.
getPaidCnt());
3808                          System.out.println("Age  (data): " + intAgeInDays.toString() +     "  (object):" + mp3.
getRawAge());
3809                          System.out.println("Free (data): " + intFreePlayedCnt.toString() + "  (object):" + mp3.
getPlayedCnt());
3810                       }
3811                    }
3812                    else
3813                    {
3814                       System.out.println("Could not find object for: " + mp3PathName);
3815                    }
3816                 }
3817              }
3818
3819              in.close();
3820           }
3821           else
3822           {
3823              System.out.println("Could not find CDStats.TXT, could not restore statistics...");
3824           }
3825        }
3826        catch (java.io.IOException ioException)
3827        {
3828           System.out.println("ERROR:  Could not read data from disk!");
3829        }
3830     }
3831
3832     /**
3833      *
3834      */
3835     private void rollBackupDataFiles(boolean bRollAll)
3836     {
3837        File bakDir1 = new File("d:/backup1");
3838
3839
3840        if (!bakDir1.exists())
3841           bakDir1.mkdir();
3842
3843
3844        // At the very least, backup the current data file to the first backup data file.
3845        try
3846        {
3847           Runtime.getRuntime().exec("CMD.EXE COPY C:\\KIOSK\\MP3Jukebox.DAT D:\\Backup1\\MP3Jukebox.DAT");
3848        }
3849        catch(java.io.IOException e)
3850        {
3851           System.out.println("Could not copy C:\\KIOSK\\MP3Jukebox.DAT to D:\\Backup1\\MP3Jukebox.DAT");
3852        }
3853
3854
3855        if (bRollAll)
3856        {
3857           File bak1    = new File("d:/backup1/MP3Jukebox.DAT");
3858
3859           File bakDir2 = new File("d:/backup2");
3860           File bak2    = new File("d:/backup2/MP3Jukebox.DAT");
3861
3862           File bakDir3 = new File("d:/backup3");
3863           File bak3    = new File("d:/backup3/MP3Jukebox.DAT");
3864
3865           File bakDir4 = new File("d:/backup4");
```

```
3866            File bak4    = new File("d:/backup4/MP3Jukebox.DAT");
3867
3868            File bakDir5 = new File("d:/backup5");
3869            File bak5    = new File("d:/backup5/MP3Jukebox.DAT");
3870
3871            File bakDir6 = new File("d:/backup6");
3872            File bak6    = new File("d:/backup6/MP3Jukebox.DAT");
3873
3874            File bakDir7 = new File("d:/backup7");
3875            File bak7    = new File("d:/backup7/MP3Jukebox.DAT");
3876
3877            // Make sure the directory structure exists.
3878            if (!bakDir7.exists())
3879               bakDir7.mkdir();
3880
3881            if (!bakDir6.exists())
3882               bakDir6.mkdir();
3883
3884            if (!bakDir5.exists())
3885               bakDir5.mkdir();
3886
3887            if (!bakDir4.exists())
3888               bakDir4.mkdir();
3889
3890            if (!bakDir3.exists())
3891               bakDir3.mkdir();
3892
3893            if (!bakDir2.exists())
3894               bakDir2.mkdir();
3895
3896
3897            // Now, roll the files.
3898            if (bak7.exists())
3899               bak7.delete();
3900
3901            if (bak6.exists())
3902               bak6.renameTo(bak7);
3903
3904            if (bak5.exists())
3905               bak5.renameTo(bak6);
3906
3907            if (bak4.exists())
3908               bak4.renameTo(bak5);
3909
3910            if (bak3.exists())
3911               bak3.renameTo(bak4);
3912
3913            if (bak2.exists())
3914               bak2.renameTo(bak3);
3915
3916            if (bak1.exists())
3917               bak1.renameTo(bak2);
3918         }
3919      }
3920
3921      public boolean setSelectedCDParentRowBySong(JTree tree, PlayListEntry mp3)
3922      {
3923         DefaultMutableTreeNode root = (DefaultMutableTreeNode)treeModel.getRoot();;
3924         DefaultMutableTreeNode tmpNode = null;
3925         DefaultMutableTreeNode parentNode = null;
3926         Object                 tmpObject;
3927         boolean                bFoundSong = false;
3928         boolean                bSuccess   = false;
3929
3930         // Perform a Prefix traversal of the tree.
3931         for (Enumeration e = root.preorderEnumeration(); e.hasMoreElements() && !bFoundSong; )
3932         {
3933            tmpNode = (DefaultMutableTreeNode)e.nextElement();
3934
3935            if (isPlayListEntry(tmpNode))
3936            {
3937               tmpObject = tmpNode.getUserObject();
3938
3939               if (mp3.getMp3Path().equalsIgnoreCase(((PlayListEntry)tmpObject).getMp3Path()))
3940               {
3941                  bFoundSong = true;
3942                  parentNode = (DefaultMutableTreeNode)tmpNode.getParent();
3943               }
3944            }
3945         }
3946
3947
3948         // Now, select the row corresponding to the CD itself.
3949         if (bFoundSong == true)
3950         {
3951            TreePath treePath = new TreePath(parentNode.getPath());
3952
3953            if (treePath == null)
3954            {
3955               System.out.println("Error getting path for song: " + mp3.toString());
```

```
3956            }
3957            else
3958            {
3959               int iRow = tree.getRowForPath(treePath);
3960               tree.setSelectionRow(iRow);
3961               bSuccess = true;
3962            }
3963         }
3964
3965         return bSuccess;
3966      }
3967   }
3968
```

ATTACHMENT 3

```
/**
 * Filename: GBAMgr.java
 *
 * Author: Tom Myers
 *
 * Version: 1.0
 *
 * Purpose: This file contains the code for the GBAMgr object class, used to
 *          to handle the communications to the VS2CK bill acceptor by means of a connection
 *          to the specified serial communications port.  This class models the state behavior
 *          of the bill acceptor and has public methods to allow the main application to
 *          query the current state of the acceptor (e.g. "Was a bill accepted?")
 *
 * Inputs: The following parameters are used to construct objects of this class:
 *          1. port - A String that tells us which serial port to use.
 *          2. debug - A boolean that toggles whether debug information is printed to System.out
 *          3. poll - An integer that specifies the polling rate to the bill acceptor
 *            (in milliseconds)
 *
 * Outputs: None
 *
 * Development Environment:  JDK 1.3 (Sun's Java 2 Standard Edition version 1.3) was used.
 *          The javax.comm package from Sun was also used to facilitate serial communications.
 *
 *
 * (c) Copyright 2000 Digital Jukebox Technologies LLC.  All Rights Reserved.
 */

import java.io.*;
import java.util.*;
import javax.comm.*;

public class GBAMgr implements Runnable, SerialPortEventListener
{
    //--For messaging.
    /** Polls the acceptor, telling it to stack the bill (uses 0 for an ack) */
    public static final int POLL_AND_STACK0  = 0;

    /** Polls the acceptor, telling it to stack the bill (uses 1 for an ack) */
    public static final int POLL_AND_STACK1  = 1;

    /** Polls the acceptor, telling it to return the bill (uses 0 for an ack) */
    public static final int POLL_AND_RETURN0 = 2;

    /** Polls the acceptor, telling it to return the bill (uses 1 for an ack) */
    public static final int POLL_AND_RETURN1 = 3;

    /** Message type for when a $1, $2, $5, $10, or $20 dollar bill is escrowed. */
    public static final int POLL_AND_STACK   = 4;

    /** Message type for when a $50 or $100 dollar bill is escrowed. */
    public static final int POLL_AND_RETURN  = 5;

    /** Message type for the master:
     * byte[] byteArray = {stx,len,msg_and_ack,data0,data1,data2,etx,checksum}
     */
    public static final int GBA_MASTER       = 6;

    /** Message type for the master:
     * byte[] byteArray = {stx,len,msg_and_ack,data0,data1,data2,data3,data4,data5,etx,checksum};
     */
    public static final int GBA_SLAVE        = 7;

    //--For acceptor states.
    /** Idle acceptor state. */
    public static final int IDLE      = 1;

    /** Accepting acceptor state. */
    public static final int ACCEPTING = 2;

    /** Escrowed acceptor state. */
    public static final int ESCROWED  = 4;

    /** Stacking acceptor state. */
    public static final int STACKING  = 8;

    /** Returning acceptor state. */
    public static final int RETURNING = 32;

    /** Jammed acceptor state. */
    public static final int JAMMED    = 512;

    /** Cashbox Full acceptor state. */
    public static final int FULL      = 1024;

    /** General failure acceptor state. */
    public static final int FAILURE   = 8192;

    //--For acceptor events.
    /** NULL acceptor event. */
```

```
91      public static final int NONE     = 0;
92
93      /** Stacked acceptor event. */
94      public static final int STACKED  = 16;
95
96      /** Returned acceptor event. */
97      public static final int RETURNED = 64;
98
99      /** Cheated acceptor event. */
100     public static final int CHEATED  = 128;
101
102     /** Rejected acceptor event. */
103     public static final int REJECTED = 256;
104
105     /** Power-up acceptor event. */
106     public static final int POWERUP  = 2048;
107
108     //--For bill denominations.
109     /** NULL bill type. */
110     public static final int UNKNOWN = 0;
111
112     /** One dollar bill type. */
113     public static final int ONE     = 1;
114
115     /** Two dollar bill type. */
116     public static final int TWO     = 2;
117
118     /** Five dollar bill type. */
119     public static final int FIVE    = 3;
120
121     /** Ten dollar bill type. */
122     public static final int TEN     = 4;
123
124     /** Twenty dollar bill type. */
125     public static final int TWENTY  = 5;
126
127     /** Fifty dollar bill type. */
128     public static final int FIFTY   = 6;
129
130     /** Hundred dollar bill type. */
131     public static final int HUNDRED = 7;
132
133
134
135     // For finding the port
136     private Enumeration portList;
137     private CommPortIdentifier portId;
138     private SerialPort serialPort;
139
140
141     // For Debugging.
142     private boolean bDebug = false;
143
144
145     // For reading.
146     private InputStream inputStream;
147
148
149     // For writing (a.k.a polling).
150     private OutputStream outputStream;
151     private Thread writeThread;
152     private int iMessageFlag;
153     private int iMessageType;
154     private int iPollInterval;
155     private GBAMessage gbaPollMessage0;
156     private GBAMessage gbaPollMessage1;
157     private GBAMessage gbaReturnMessage0;
158     private GBAMessage gbaReturnMessage1;
159
160
161     // For error handling;
162     private int iRet;
163
164
165     // GBA States (IDLE-RETURNING contained in data0 byte from Slave).
166     // Note:  states are continuous, may be present in multiple messages.
167     private int currentState = IDLE;
168
169
170     // GBA Events (only reported ONCE to the Master)
171     private int currentEvent = NONE;
172     private int lastEvent = NONE;
173
174
175     // Other Message Information (not a state or event)
176     private boolean bCashBoxPresent = true;
177
178
179     // Bill demoninations.
180     private int billValue = UNKNOWN;
```

```
181       private int lastBillProcessed = UNKNOWN;
182
183
184       // ##########################################################################
185       // PUBLIC METHODS
186       // ##########################################################################
187
188       /**
189        * Returns the current state of the acceptor.
190        * @return    One of the following strings:
191        * <pre>
192        * IDLE
193        * ACCEPTING
194        * ESCROWED
195        * STACKING
196        * RETURNING
197        * JAMMED
198        * FULL
199        * FAILURE
200        * </pre>
201        */
202       synchronized public String getCurrentState()
203       {
204          String strCurrentState;
205
206          switch (currentState)
207          {
208             case IDLE:
209                strCurrentState = "IDLE";
210                break;
211
212             case ACCEPTING:
213                strCurrentState = "ACCEPTING";
214                break;
215
216             case ESCROWED:
217                strCurrentState = "ESCROWED";
218                break;
219
220             case STACKING:
221                strCurrentState = "STACKING";
222                break;
223
224             case RETURNING:
225                strCurrentState = "RETURNING";
226                break;
227
228             case JAMMED:
229                strCurrentState = "JAMMED";
230                break;
231
232             case FULL:
233                strCurrentState = "FULL";
234                break;
235
236             case FAILURE:
237                strCurrentState = "FAILURE";
238
239             default:
240                strCurrentState = "CANNOT DETERMINE STATE";
241          }
242
243          if (bDebug)
244             System.out.println("GBAMGR: Current state: " + strCurrentState);
245
246          return strCurrentState;
247       }
248
249       /**
250        * Returns the last event processed by the acceptor.
251        @ return    One of the following strings:
252        * <pre>
253        * NONE (or no events processed since the last event, or since powerup)
254        * STACKED
255        * RETURNED
256        * CHEATED
257        * REJECTED
258        * POWERUP
259        * </pre>
260        */
261       synchronized public String getLastEvent()
262       {
263          String strLastEvent;
264
265          switch (lastEvent)
266          {
267             case NONE:
268                strLastEvent = "NONE";
269                break;
270
```

```
271             case STACKED:
272                strLastEvent = "STACKED";
273                break;
274
275             case RETURNED:
276                strLastEvent = "RETURNED";
277                break;
278
279             case CHEATED:
280                strLastEvent = "CHEATED";
281                break;
282
283             case REJECTED:
284                strLastEvent = "REJECTED";
285                break;
286
287             case POWERUP:
288                strLastEvent = "POWERUP";
289                break;
290
291             default:
292                strLastEvent = "CANNOT DETERMINE LAST STATE";
293          }
294
295          if (bDebug)
296             System.out.println("GBAMGR: Last Event Encountered: " + strLastEvent);
297
298          resetLastEvent();
299
300          return strLastEvent;
301       }
302
303       /**
304        * Resets the last event field to NONE.
305        */
306       synchronized public void resetLastEvent()
307       {
308          if (bDebug)
309             System.out.println("GBAMGR: Resetting Last Event to: NONE");
310
311          lastEvent = NONE;
312       }
313
314       /**
315        * Returns the last bill processed by the acceptor (since the last query), which is one of the following:
316        * @return    One of the following strings:
317        * <pre>
318        * UNKNOWN (if no bill as been processed since the last call to this method, or since startup.)
319        * ONE
320        * TWO
321        * FIVE
322        * TEN
323        * TWENTY
324        * FIFTY
325        * HUNDRED
326        * </pre>
327        */
328       synchronized public String getLastBillProcessed()
329       {
330          String strLastBillProcessed;
331
332          switch (lastBillProcessed)
333          {
334             case UNKNOWN:
335                strLastBillProcessed = "UNKNOWN";
336                break;
337
338             case ONE:
339                strLastBillProcessed = "ONE";
340                break;
341
342             case TWO:
343                strLastBillProcessed = "TWO";
344                break;
345
346             case FIVE:
347                strLastBillProcessed = "FIVE";
348                break;
349
350             case TEN:
351                strLastBillProcessed = "TEN";
352                break;
353
354             case TWENTY:
355                strLastBillProcessed = "TWENTY";
356                break;
357
358             case FIFTY:
359                strLastBillProcessed = "FIFTY";
360                break;
```

```
            case HUNDRED:
               strLastBillProcessed = "HUNDRED";
               break;

            default:
               strLastBillProcessed = "CANNOT DETERMINE LAST BILL PROCESSED";
         }

         if (bDebug)
            System.out.println("GBAMGR: Last Bill Processed: " + strLastBillProcessed);

         resetLastBillProcessed();

         return strLastBillProcessed;
      }

      /**
       * Resets the last bill processed (since the last query) to UNKNOWN.
       */
      synchronized public void resetLastBillProcessed()
      {
         if (bDebug)
            System.out.println("GBAMGR: Resetting Last Bill to: UNKNOWN");

         lastBillProcessed = UNKNOWN;
      }

      /**
       * Returns the flag corresponding to whether the cashbox is present or not.
       * @return <pre>true</pre> if the cashbox is present, <pre>false</pre> otherwise.
       */
      synchronized public boolean isCashBoxPresent()
      {
         if (bDebug)
            System.out.println("GBAMGR: Cash Box Present: " + bDebug);

         return bCashBoxPresent;
      }

      /**
       * Returns the last initialization return code. note: Use the debug=true constructor flag instead.
       * @return    iRet the last return code set during initialization.
       */
      synchronized public int getLastInitReturnCode()
      {
         if (bDebug)
            System.out.println("GBAMGR: Last Init Return Code: " + iRet);

         return iRet;
      }

      /**
       * Default constructor, using port=COM2, debug=false, and poll=500 as defaults.
       */
      public GBAMgr()
      {
        this("COM2", false, 500);
      }

      /**
       * Uses the specified port and the defaults of debug=false and poll=500.
       * @param    port    The port to use.
       */
      public GBAMgr(String port)
      {
         this(port, false, 500);
      }

      /**
       * Uses the specified debug flag and the defaults of port=COM2 and poll=500.
       * @param    debug   The debug flag to use.
       */
      public GBAMgr(boolean debug)
      {
         this("COM2", debug, 500);
      }

      /**
       * Uses the specified polling rate and the defaults of port=COM2 and debug=false.
       * @param    poll    The polling rate to use (in milliseconds).
       */
      public GBAMgr(int poll)
      {
         this("COM2", false, poll);
      }

      /**
       * Uses the specified port and debug flag and the default of poll=500.
       * @param    port    The port to use.
       * @param    debug   The debug flag to use.
```

```
452         */
453        public GBAMgr(String port, boolean debug)
454        {
455           this(port, debug, 500);
456        }
457
458        /**
459         * Uses the specified port and polling rate and the default of debug=false.
460         * @param    port    The port to use.
461         * @param    poll    The polling rate to use (in milliseconds).
462         */
463        public GBAMgr(String port, int poll)
464        {
465           this(port, false, poll);
466        }
467
468        /**
469         * Uses the specified debug flag and polling rate and the default of port=COM2.
470         * @param    debug   The debug flag to use.
471         * @param    poll    The polling rate to use (in milliseconds).
472         */
473        public GBAMgr(boolean debug, int poll)
474        {
475           this("COM2", debug, poll);
476        }
477
478        /**
479         * Uses the specified port, debug flag, and polling rate.
480         * @param    port    The port to use.
481         * @param    debug   The debug flag to use.
482         * @param    poll    The polling rate to use (in milliseconds).
483         */
484        public GBAMgr(String port, boolean debug, int poll)
485        {
486           if (bDebug)
487           {
488              System.out.println("GBAMGR: port=  " + port);
489              System.out.println("GBAMGR: debug= " + debug);
490              System.out.println("GBAMGR: poll=  " + poll);
491           }
492
493           // Set the polling rate (in milliseconds).
494           if (poll > 0  && poll <= 5000 )
495              iPollInterval = poll;
496           else
497           {
498              System.out.println("GBAMGR: Invalid poll interval.  Using 500ms instead.");
499              iPollInterval = 500;
500           }
501
502           // Turn on System.out.println debug statements.
503           bDebug = debug;
504
505           // Initialize data members.
506           iMessageFlag = 0;
507           iMessageType = POLL_AND_STACK;
508           iRet = -99;
509
510
511           // Initialize the messages to be sent to the acceptor.
512           gbaPollMessage0 = new GBAMessage(POLL_AND_STACK0);
513           gbaPollMessage1 = new GBAMessage(POLL_AND_STACK1);
514           gbaReturnMessage0 = new GBAMessage(POLL_AND_RETURN0);
515           gbaReturnMessage1 = new GBAMessage(POLL_AND_RETURN1);
516
517
518           // Open the port for read/write.
519           portList = CommPortIdentifier.getPortIdentifiers();
520           while (portList.hasMoreElements())
521           {
522              portId = (CommPortIdentifier) portList.nextElement();
523
524              if (portId.getPortType() == CommPortIdentifier.PORT_SERIAL)
525              {
526                 if (portId.getName().equalsIgnoreCase(port))
527                 {
528                    if (bDebug)
529                       System.out.println("GBAMGR: Found port: " + port);
530
531                    iRet = 0;
532
533                    try
534                    {
535                       serialPort = (SerialPort)portId.open("GBAMgr", 2000);
536                       if (bDebug)
537                          System.out.println("GBAMGR: Opening port: " + port);
538                    }
539                    catch (PortInUseException e)
540                    {
541                       System.out.println("GBAMGR: Port in Use!!!");
```

```
542                  System.out.println("GBAMGR:  ");
543                  e.printStackTrace();
544                  iRet = -1;
545               }
546
547               try
548               {
549                  outputStream = serialPort.getOutputStream();
550                  if (bDebug)
551                     System.out.println("GBAMGR: Getting output stream");
552               }
553               catch (IOException e)
554               {
555                  System.out.println("GBAMGR: Can't get output stream!!!");
556                  System.out.println("GBAMGR:  ");
557                  e.printStackTrace();
558                  iRet = -2;
559               }
560
561               try
562               {
563                  inputStream = serialPort.getInputStream();
564                  if (bDebug)
565                     System.out.println("GBAMGR: Getting inputStream");
566               }
567               catch (IOException e)
568               {
569                  System.out.println("GBAMGR: Could not get inputStream!!!");
570                  System.out.println("GBAMGR:  ");
571                  e.printStackTrace();
572                  iRet = -3;
573               }
574
575               try
576               {
577                  serialPort.setSerialPortParams(9600,
578                                                 SerialPort.DATABITS_7,
579                                                 SerialPort.STOPBITS_1,
580                                                 SerialPort.PARITY_EVEN);
581                  if (bDebug)
582                  {
583                     System.out.println("GBAMGR: Setting serial port parameters:");
584                     System.out.println("GBAMGR:      SPEED     = 9600");
585                     System.out.println("GBAMGR:      DATA_BITS = 7");
586                     System.out.println("GBAMGR:      STOP_BITS = 1");
587                     System.out.println("GBAMGR:      PARITY    = EVEN");
588                  }
589               }
590               catch (UnsupportedCommOperationException e)
591               {
592                  System.out.println("GBAMGR: Could not set serial port parameters!!!");
593                  System.out.println("GBAMGR:  ");
594                  e.printStackTrace();
595                  iRet = -4;
596               }
597
598               try
599               {
600                  serialPort.addEventListener(this);
601                  if (bDebug)
602                     System.out.println("GBAMGR: Adding serial port event listener");
603               }
604               catch (TooManyListenersException e)
605               {
606                  System.out.println("GBAMGR: Too many serial port listeners!!!");
607                  System.out.println("GBAMGR:  ");
608                  e.printStackTrace();
609                  iRet = -5;
610               }
611
612               serialPort.notifyOnDataAvailable(true);
613
614               if (iRet == 0)
615               {
616                  try
617                  {
618                     Thread.sleep(2000);
619                  }
620                  catch (InterruptedException e)
621                  {
622                     System.out.println("GBAMGR: Could not go to sleep!!!");
623                     System.out.println("GBAMGR:  ");
624                     e.printStackTrace();
625                     iRet = -8;
626                  }
627               }
628
629               if (iRet == 0)
630               {
631                  writeThread = new Thread(this);
```

```
                writeThread.start();
                if (bDebug)
                   System.out.println("GBAMGR: Starting poller thread, interval: " + iPollInterval);
             }
          }
       }
    }
}
// ############################################################################


// PRIVATE METHODS: ##########################################################
private class GBAMessage
{
   private byte stx;
   private byte len;
   private byte msg_and_ack;
   private byte data0;
   private byte data1;
   private byte data2;
   private byte data3;
   private byte data4;
   private byte data5;
   private byte etx;
   private byte checksum;

   public GBAMessage(int messageType)
   {
      switch (messageType)
      {
         case POLL_AND_STACK0:
            stx = stringToHexByte("02");
            len = stringToHexByte("08");
            msg_and_ack = stringToHexByte("10");
            data0 = stringToHexByte("1F");
            data1 = stringToHexByte("31"); // Stack bill
            data2 = stringToHexByte("00");
            etx = stringToHexByte("03");
            checksum = calculateChecksum(len, msg_and_ack, data0, data1, data2);
         break;

         case POLL_AND_STACK1:
            stx = stringToHexByte("02");
            len = stringToHexByte("08");
            msg_and_ack = stringToHexByte("11");
            data0 = stringToHexByte("1F");
            data1 = stringToHexByte("31");  // Stack bill
            data2 = stringToHexByte("00");
            etx = stringToHexByte("03");
            checksum = calculateChecksum(len, msg_and_ack, data0, data1, data2);
         break;

         case POLL_AND_RETURN0:
            stx = stringToHexByte("02");
            len = stringToHexByte("08");
            msg_and_ack = stringToHexByte("10");
            data0 = stringToHexByte("1F");
            data1 = stringToHexByte("51"); // Return bill
            data2 = stringToHexByte("00");
            etx = stringToHexByte("03");
            checksum = calculateChecksum(len, msg_and_ack, data0, data1, data2);
         break;

         case POLL_AND_RETURN1:
            stx = stringToHexByte("02");
            len = stringToHexByte("08");
            msg_and_ack = stringToHexByte("11");
            data0 = stringToHexByte("1F");
            data1 = stringToHexByte("51"); // Return bill
            data2 = stringToHexByte("00");
            etx = stringToHexByte("03");
            checksum = calculateChecksum(len, msg_and_ack, data0, data1, data2);
         break;
      }
   }

   private byte calculateChecksum(byte b1, byte b2, byte b3, byte b4, byte b5)
   {
      byte byteVal = b1 ^= b2 ^= b3 ^= b4 ^= b5;

      return byteVal;
   }

   private byte getMasterMsgByte(int index)
   {
     byte byteVal = 0;
     switch (index)
     {
        case 0:
           byteVal = stx;
```

```
722                 break;
723              case 1:
724                 byteVal = len;
725                 break;
726              case 2:
727                 byteVal = msg_and_ack;
728                 break;
729              case 3:
730                 byteVal = data0;
731                 break;
732              case 4:
733                 byteVal = data1;
734                 break;
735              case 5:
736                 byteVal = data2;
737                 break;
738              case 6:
739                 byteVal = etx;
740                 break;
741              case 7:
742                 byteVal = checksum;
743                 break;
744           }
745           return byteVal;
746        }
747
748        private byte[] getByteArray(int msgType)
749        {
750           if (msgType == GBA_MASTER)
751           {
752              byte[] byteArray = {stx,len,msg_and_ack,data0,data1,data2,etx,checksum};
753              return byteArray;
754           }
755           else
756           {
757              byte[] byteArray = {stx,len,msg_and_ack,data0,data1,data2,data3,data4,data5,etx,checksum};
758              return byteArray;
759           }
760        }
761
762        private byte stringToHexByte(String s)
763        {
764           byte byteVal = 0;
765
766           try
767           {
768              byteVal = Byte.parseByte(s, 16);
769           }
770           catch (java.lang.NumberFormatException e)
771           {
772              System.out.println("GBAMGR: Couldn't format byte: " + s + " (hex)");
773           }
774
775           return byteVal;
776        }
777     }
778
779     private int write(GBAMessage gbaMessage)
780     {
781        int iReturn = 0;
782        byte[] byteArray = gbaMessage.getByteArray(GBA_MASTER);
783        String strMessage = new String(byteArray);
784
785        try
786        {
787           outputStream.write(strMessage.getBytes());
788
789           if (bDebug)
790           {
791              System.out.print("SENT:     ");
792              System.out.print(" " + formatByte(gbaMessage.getMasterMsgByte(0)));
793              System.out.print(" " + formatByte(gbaMessage.getMasterMsgByte(1)));
794              System.out.print(" " + formatByte(gbaMessage.getMasterMsgByte(2)));
795              System.out.print(" " + formatByte(gbaMessage.getMasterMsgByte(3)));
796              System.out.print(" " + formatByte(gbaMessage.getMasterMsgByte(4)));
797              System.out.print(" " + formatByte(gbaMessage.getMasterMsgByte(5)));
798              System.out.print(" " + formatByte(gbaMessage.getMasterMsgByte(6)));
799              System.out.print(" " + formatByte(gbaMessage.getMasterMsgByte(7)));
800              System.out.print("\n");
801           }
802        }
803        catch (IOException e)
804        {
805           System.out.println("GBAMGR: Couldn't write to port!!!");
806           System.out.println("GBAMGR: Data: " + strMessage);
807           System.out.println("GBAMGR:  ");
808           e.printStackTrace();
809           iReturn = -1;
810        }
811
```

```
812         return iReturn;
813      }
814
815      public void serialEvent(SerialPortEvent event)
816      {
817         int numBytes = 0;
818
819         switch(event.getEventType())
820         {
821            case SerialPortEvent.BI:
822            case SerialPortEvent.OE:
823            case SerialPortEvent.FE:
824            case SerialPortEvent.PE:
825            case SerialPortEvent.CD:
826            case SerialPortEvent.CTS:
827            case SerialPortEvent.DSR:
828            case SerialPortEvent.RI:
829            case SerialPortEvent.OUTPUT_BUFFER_EMPTY:
830               break;
831
832            case SerialPortEvent.DATA_AVAILABLE:
833               numBytes = 0;
834               byte[] readBuffer = new byte[32];
835
836               try
837               {
838                  while (inputStream.available() > 0)
839                  {
840                     numBytes = inputStream.read(readBuffer);
841                  }
842
843                  // If the message is 11 bytes:
844                  //   0    1           2      3      4      5      6      7      8    9        10
845                  // STX, LEN, MSGTYPE&ACK, Data0, Data1, Data2, Data3, Data4, Data5, ETX, Checksum
846                  // Then, decode it...
847                  if (readBuffer[0] == 2)
848                  {
849                     decodeGBAMessage(readBuffer);
850                  }
851               }
852               catch (IOException e)
853               {
854                  System.out.println("GBAMGR: Could not read from the port!!!");
855                  System.out.println("GBAMGR:  ");
856                  e.printStackTrace();
857                  iRet = -7;
858               }
859               break;
860         }
861      }
862
863      /**
864       *   Message Format: (for each byte)
865       *  STX, LEN, MSGTYPE&ACK, Data0, Data1, Data2, Data3, Data4, Data5, ETX, Checksum
866       *
867       */
868      public void decodeGBAMessage(byte[] readBuffer)
869      {
870         // Determine the current state.
871         if (isBitSet(readBuffer[3], 0))
872            currentState = IDLE;
873         else if (isBitSet(readBuffer[3], 1))
874            currentState = ACCEPTING;
875         else if (isBitSet(readBuffer[3], 2))
876            currentState = ESCROWED;
877         else if (isBitSet(readBuffer[3], 3))
878            currentState = STACKING;
879         else if (isBitSet(readBuffer[3], 5))
880            currentState = RETURNING;
881         else if (isBitSet(readBuffer[4], 2))
882            currentState = JAMMED;
883         else if (isBitSet(readBuffer[4], 3))
884            currentState = FULL;
885         else if (isBitSet(readBuffer[5], 2))
886            currentState = FAILURE;
887
888
889         // Determine the bill value (if any is presently being processed).
890         billValue = decodeBillType(readBuffer[5]);
891         if (billValue != UNKNOWN)
892            lastBillProcessed = billValue;
893
894
895         // Determine if an event has happened.
896         if (isBitSet(readBuffer[3], 4))
897            currentEvent = STACKED;
898         else if (isBitSet(readBuffer[3], 6))
899            currentEvent = RETURNED;
900         else if (isBitSet(readBuffer[4], 0))
901            currentEvent = CHEATED;
```

```
902          else if (isBitSet(readBuffer[4], 1))
903             currentEvent = REJECTED;
904          else if (isBitSet(readBuffer[5], 0))
905             currentEvent = POWERUP;
906          else
907             currentEvent = NONE;
908
909          if (currentEvent != NONE)
910             lastEvent = currentEvent;
911
912
913          // Determine if the LCB is present.
914          if (isBitSet(readBuffer[4], 4))
915             bCashBoxPresent = true;
916          else
917             bCashBoxPresent = false;
918
919
920          // Display the message.
921          if (bDebug)
922          {
923             System.out.print("RECEIVED: ");
924
925             System.out.print(" " + formatByte(readBuffer[3]));
926             System.out.print(" " + formatByte(readBuffer[4]));
927             System.out.print(" " + formatByte(readBuffer[5]));
928
929             System.out.print(" " + "State: " + getCurrentState());
930             System.out.print(" " + "Event: " + getLastEvent());
931             System.out.print(" " + "Value: " + getLastBillProcessed());
932             System.out.print(" " + "LCB: " + bCashBoxPresent);
933
934             System.out.print("\n");
935          }
936
937
938          // If a $50 or $100 bill was escrowed, then return it to the user.
939          if (currentState == ESCROWED && (billValue == FIFTY || billValue == HUNDRED))
940          {
941             setMessageType(POLL_AND_RETURN);
942          }
943       }
944
945       synchronized private void setMessageType(int type)
946       {
947          iMessageType = type;
948       }
949
950       private boolean isBitSet(byte byteVal, int bit)
951       {
952          boolean bBitSet = false;
953
954          byte mask   = 0;
955          byte result = 0;
956
957          switch (bit)
958          {
959             case 0:
960                mask = 1;
961                break;
962             case 1:
963                mask = 2;
964                break;
965             case 2:
966                mask = 4;
967                break;
968             case 3:
969                mask = 8;
970                break;
971             case 4:
972                mask = 16;
973                break;
974             case 5:
975                mask = 32;
976                break;
977             case 6:
978                mask = 64;
979                break;
980          }
981
982          result = byteVal &= mask;
983
984          if (result != 0)
985             bBitSet = true;
986
987          return bBitSet;
988       }
989
990       private int decodeBillType(int mask)
991       {
```

```
992          int iBill = UNKNOWN;
993
994          if (mask >= 8 && mask <= 15)
995             iBill = ONE;
996          else if (mask >= 16 && mask <= 23)
997             iBill = TWO;
998          else if (mask >= 24 && mask <= 31)
999             iBill = FIVE;
1000         else if (mask >= 32 && mask <= 39)
1001            iBill = TEN;
1002         else if (mask >= 40 && mask <= 47)
1003            iBill = TWENTY;
1004         else if (mask >= 48 && mask <= 55)
1005            iBill = FIFTY;
1006         else if (mask >= 56 && mask <= 63)
1007            iBill = HUNDRED;
1008
1009         return iBill;
1010      }
1011
1012      private String formatByte(byte b)
1013      {
1014         String val = null;
1015
1016         switch (b)
1017         {
1018            case 0:
1019               val = "00";
1020               break;
1021            case 1:
1022               val = "01";
1023               break;
1024            case 2:
1025               val = "02";
1026               break;
1027            case 3:
1028               val = "03";
1029               break;
1030            case 4:
1031               val = "04";
1032               break;
1033            case 5:
1034               val = "05";
1035               break;
1036            case 6:
1037               val = "06";
1038               break;
1039            case 7:
1040               val = "07";
1041               break;
1042            case 8:
1043               val = "08";
1044               break;
1045            case 9:
1046               val = "09";
1047               break;
1048            case 10:
1049               val = "0A";
1050               break;
1051            case 11:
1052               val = "0B";
1053               break;
1054            case 12:
1055               val = "0C";
1056               break;
1057            case 13:
1058               val = "0D";
1059               break;
1060            case 14:
1061               val = "0E";
1062               break;
1063            case 15:
1064               val = "0F";
1065               break;
1066
1067            case 16:
1068               val = "10";
1069               break;
1070            case 17:
1071               val = "11";
1072               break;
1073            case 18:
1074               val = "12";
1075               break;
1076            case 19:
1077               val = "13";
1078               break;
1079            case 20:
1080               val = "14";
1081               break;
```

```
1082            case 21:
1083               val = "15";
1084               break;
1085            case 22:
1086               val = "16";
1087               break;
1088            case 23:
1089               val = "17";
1090               break;
1091            case 24:
1092               val = "18";
1093               break;
1094            case 25:
1095               val = "19";
1096               break;
1097            case 26:
1098               val = "1A";
1099               break;
1100            case 27:
1101               val = "1B";
1102               break;
1103            case 28:
1104               val = "1C";
1105               break;
1106            case 29:
1107               val = "1D";
1108               break;
1109            case 30:
1110               val = "1E";
1111               break;
1112            case 31:
1113               val = "1F";
1114               break;
1115
1116            case 32:
1117               val = "20";
1118               break;
1119            case 33:
1120               val = "21";
1121               break;
1122            case 34:
1123               val = "22";
1124               break;
1125            case 35:
1126               val = "23";
1127               break;
1128            case 36:
1129               val = "24";
1130               break;
1131            case 37:
1132               val = "25";
1133               break;
1134            case 38:
1135               val = "26";
1136               break;
1137            case 39:
1138               val = "27";
1139               break;
1140            case 40:
1141               val = "28";
1142               break;
1143            case 41:
1144               val = "29";
1145               break;
1146            case 42:
1147               val = "2A";
1148               break;
1149            case 43:
1150               val = "2B";
1151               break;
1152            case 44:
1153               val = "2C";
1154               break;
1155            case 45:
1156               val = "2D";
1157               break;
1158            case 46:
1159               val = "2E";
1160               break;
1161            case 47:
1162               val = "2F";
1163               break;
1164
1165            case 48:
1166               val = "30";
1167               break;
1168            case 49:
1169               val = "31";
1170               break;
1171            case 50:
```

```
1172              val = "32";
1173              break;
1174           case 51:
1175              val = "33";
1176              break;
1177           case 52:
1178              val = "34";
1179              break;
1180           case 53:
1181              val = "35";
1182              break;
1183           case 54:
1184              val = "36";
1185              break;
1186           case 55:
1187              val = "37";
1188              break;
1189           case 56:
1190              val = "38";
1191              break;
1192           case 57:
1193              val = "39";
1194              break;
1195           case 58:
1196              val = "3A";
1197              break;
1198           case 59:
1199              val = "3B";
1200              break;
1201           case 60:
1202              val = "3C";
1203              break;
1204           case 61:
1205              val = "3D";
1206              break;
1207           case 62:
1208              val = "3E";
1209              break;
1210           case 63:
1211              val = "3F";
1212              break;
1213
1214           case 64:
1215              val = "40";
1216              break;
1217           case 65:
1218              val = "41";
1219              break;
1220           case 66:
1221              val = "42";
1222              break;
1223           case 67:
1224              val = "43";
1225              break;
1226           case 68:
1227              val = "44";
1228              break;
1229           case 69:
1230              val = "45";
1231              break;
1232           case 70:
1233              val = "46";
1234              break;
1235           case 71:
1236              val = "47";
1237              break;
1238           case 72:
1239              val = "48";
1240              break;
1241           case 73:
1242              val = "49";
1243              break;
1244           case 74:
1245              val = "4A";
1246              break;
1247           case 75:
1248              val = "4B";
1249              break;
1250           case 76:
1251              val = "4C";
1252              break;
1253           case 77:
1254              val = "4D";
1255              break;
1256           case 78:
1257              val = "4E";
1258              break;
1259           case 79:
1260              val = "4F";
1261              break;
```

```
1263            case 80:
1264               val = "50";
1265               break;
1266            case 81:
1267               val = "51";
1268               break;
1269            case 82:
1270               val = "52";
1271               break;
1272            case 83:
1273               val = "53";
1274               break;
1275            case 84:
1276               val = "54";
1277               break;
1278            case 85:
1279               val = "55";
1280               break;
1281            case 86:
1282               val = "56";
1283               break;
1284            case 87:
1285               val = "57";
1286               break;
1287            case 88:
1288               val = "58";
1289               break;
1290            case 89:
1291               val = "59";
1292               break;
1293            case 90:
1294               val = "5A";
1295               break;
1296            case 91:
1297               val = "5B";
1298               break;
1299            case 92:
1300               val = "5C";
1301               break;
1302            case 93:
1303               val = "5D";
1304               break;
1305            case 94:
1306               val = "5E";
1307               break;
1308            case 95:
1309               val = "5F";
1310               break;
1311
1312            case 96:
1313               val = "60";
1314               break;
1315            case 97:
1316               val = "61";
1317               break;
1318            case 98:
1319               val = "62";
1320               break;
1321            case 99:
1322               val = "63";
1323               break;
1324            case 100:
1325               val = "64";
1326               break;
1327            case 101:
1328               val = "65";
1329               break;
1330            case 102:
1331               val = "66";
1332               break;
1333            case 103:
1334               val = "67";
1335               break;
1336            case 104:
1337               val = "68";
1338               break;
1339            case 105:
1340               val = "69";
1341               break;
1342            case 106:
1343               val = "6A";
1344               break;
1345            case 107:
1346               val = "6B";
1347               break;
1348            case 108:
1349               val = "6C";
1350               break;
1351            case 109:
1352               val = "6D";
```

```
            break;
         case 110:
            val = "6E";
            break;
         case 111:
            val = "6F";
            break;

         case 112:
            val = "70";
            break;
         case 113:
            val = "71";
            break;
         case 114:
            val = "72";
            break;
         case 115:
            val = "73";
            break;
         case 116:
            val = "74";
            break;
         case 117:
            val = "75";
            break;
         case 118:
            val = "76";
            break;
         case 119:
            val = "77";
            break;
         case 120:
            val = "78";
            break;
         case 121:
            val = "79";
            break;
         case 122:
            val = "7A";
            break;
         case 123:
            val = "7B";
            break;
         case 124:
            val = "7C";
            break;
         case 125:
            val = "7D";
            break;
         case 126:
            val = "7E";
            break;
         case 127:
            val = "7F";
            break;

         default:
            val = "xx";
            break;
      }
      return val;
   }

   public void run()
   {
      int iReturnCode = 0;

      while (true)
      {
         try
         {
            Thread.sleep(iPollInterval);
         }
         catch (InterruptedException e)
         {
            System.out.println("GBAMGR: Poller: Could not go to sleep!!!");
            System.out.println("GBAMGR:  ");
            e.printStackTrace();
         }

         if (iMessageType == POLL_AND_STACK)
         {
            // If a 1,2,5,10, or 20 dollar bill was escrowed, then stack it.
            if (iMessageFlag == 0)
            {
               iReturnCode = write(gbaPollMessage0);
               iMessageFlag = 1;
            }
            else
```

```
            {
               iReturnCode = write(gbaPollMessage1);
               iMessageFlag = 0;
            }
         }
         else
         {
            // Return the bill to the user, the denomination is too high.
            if (iMessageFlag == 0)
            {
               iReturnCode = write(gbaReturnMessage0);
               iMessageFlag = 1;
            }
            else
            {
               iReturnCode = write(gbaReturnMessage1);
               iMessageFlag = 0;
            }

            // Then, resume stacking for valid bills.
            iMessageType = POLL_AND_STACK;
         }

         // Check the return code.
         if (iReturnCode != 0)
         {
            System.out.println("GBAMGR: Unsuccessful write: RC=" + iReturnCode);
         }
      }
   }


   public static void main(String[] args)
   {
      //GBAMgr gbaMgr = new GBAMgr();
      //GBAMgr gbaMgr = new GBAMgr("COM2");
      //GBAMgr gbaMgr = new GBAMgr(true);
      //GBAMgr gbaMgr = new GBAMgr("COM2", true);
      GBAMgr gbaMgr = new GBAMgr();

      int iInitReturnCode = gbaMgr.getLastInitReturnCode();
      if (iInitReturnCode != 0)
      {
         System.out.println("GBAMGR: Unsuccessful initialization: " + iInitReturnCode);
      }
   }
}
```

ATTACHMENT 4

```
/**
 * Filename: PlayerMgr.java
 *
 * Author: Tom Myers
 *
 * Version: 1.0
 *
 * Purpose: This file contains the code for the PlayerMgr object class, an abstract class used
 *          to specify the basic operations related to playing, pausing, stopping, and querying
 *          whether or not a song is playing and how much playing time is remaining for a song
 *          if it is playing.
 *
 *          This class also has methods and data members for maintaining the song queue and
 *          allowing the main application to add/remove songs to/from the song queue and to
 *          query which is the currently playing song (if any).
 *
 * Inputs: None
 *
 * Outputs: None
 *
 * Development Environment:  JDK 1.3 (Sun's Java 2 Standard Edition version 1.3) was used.
 *
 *
 * (c) Copyright 2000 Digital Jukebox Technologies LLC.  All Rights Reserved.
 */

import java.util.Vector;
import java.util.Enumeration;
import java.io.*;

import TreeMgr.*;

public abstract class PlayerMgr extends Thread
{
   public final static int PLAYING = 1;
   public final static int STOPPED = 0;
   public final static int PAUSED  = 3;

   protected Vector  playListVector;
   protected int     iStatus;
   protected int     iSongLength;
   protected int     iTimeRemaining;
   protected String  strCurrentSong;
   protected boolean bLockOnQueue;
   protected int     iNumToQueue;
   protected int     iVolume;
   protected boolean bFirstTime;
   protected boolean bIsCurrSongFree;
   protected PlayListEntry currentPlayListObj;

   public PlayerMgr()
   {
       playListVector  = new Vector(0);
       iStatus         = STOPPED;
       strCurrentSong  = new String("");
       bLockOnQueue    = false;
       iSongLength     = 0;
       iTimeRemaining  = 0;
       bFirstTime      = true;
       bIsCurrSongFree = false;
       iNumToQueue     = 5;
       iVolume         = 60;
   }

   public abstract void setVolume(int iVol);

   public abstract int getOutputTime();

   public abstract int getSongLength();

   public abstract void pressPause();

   public abstract void pressStop();

   public abstract void cleanUp();

   public abstract void play(PlayListEntry mp3);

   public abstract void run();


   public int getStatus()
   {
      return iStatus;
   }

   public boolean getInitialLock()
   {
      return bFirstTime;
   }
```

```
92        public void setInitialLock(boolean bFirst)
93        {
94           bFirstTime = bFirst;
95        }
96
97        public void releaseInitialLock()
98        {
99           if (bFirstTime)
100             bFirstTime = false;
101       }
102
103
104       public int getVolume()
105       {
106          return iVolume;
107       }
108
109       synchronized public void setLockOnQueue(boolean bFlag)
110       {
111          bLockOnQueue = bFlag;
112       }
113
114       synchronized public boolean getLockOnQueue()
115       {
116          return bLockOnQueue;
117       }
118
119       public boolean isCurrentSongFree()
120       {
121          return bIsCurrSongFree;
122       }
123
124       public Vector loadPlayList(File file)
125       {
126          Vector vector = new Vector();
127          boolean bDone = false;
128          String mp3    = null;
129
130          try
131          {
132             BufferedReader in = new BufferedReader(new FileReader(file));
133             File   tmpFile = null;
134
135             while (!bDone)
136             {
137                mp3 = in.readLine();
138                if (mp3 != null)
139                {
140                   tmpFile = new File(mp3);
141
142                   if (tmpFile.exists())
143                      vector.addElement(mp3);
144                }
145                else
146                   bDone = true;
147             }
148
149             in.close();
150             in = null;
151          }
152          catch (IOException e)
153          {
154             e.printStackTrace();
155          }
156
157          return vector;
158       }
159
160       public void savePlayList(File file) throws java.io.FileNotFoundException
161       {
162          try
163          {
164             PrintWriter out = new PrintWriter(new BufferedWriter(new FileWriter(file)));
165
166             for (Enumeration e = playListVector.elements(); e.hasMoreElements(); )
167             {
168                out.println(((PlayListEntry)e.nextElement()).getMp3Path());
169             }
170             out.flush();
171             out.close();
172             out = null;
173          }
174          catch (java.io.FileNotFoundException e1)
175          {
176             throw e1;
177          }
178          catch (IOException e)
179          {
180             e.printStackTrace();
181          }
```

```
182     }
183
184     public String getCurrentSong()
185     {
186        return strCurrentSong;
187     }
188
189     public PlayListEntry getCurrentPlayListObject()
190     {
191        return currentPlayListObj;
192     }
193
194     public PlayListEntry getNextPlayListObject()
195     {
196        PlayListEntry mp3 = null;
197
198        if (playListVector.size() > 0)
199        {
200           mp3 = ((PlayListEntry)playListVector.firstElement());
201        }
202
203        return mp3;
204     }
205
206     public boolean isReadyForNextMp3()
207     {
208        if ( (playListVector.size() < iNumToQueue) && (!bLockOnQueue) )
209           return true;
210        else
211           return false;
212     }
213
214     public int getQueuedSongCount()
215     {
216        return playListVector.size();
217     }
218
219     public void setNumberToQueue(int iNum)
220     {
221        iNumToQueue = iNum;
222     }
223
224     public int getNumberToQueue()
225     {
226        return iNumToQueue;
227     }
228
229     public Vector getPlayListVector()
230     {
231        return playListVector;
232     }
233
234     public void setPlayListVector(Vector vect)
235     {
236        playListVector = vect;
237     }
238
239     public void addPaidSongToPlayList(PlayListEntry mp3)
240     {
241        mp3.incrementQueuedCnt();
242        mp3.incrementPaidQueuedCnt();
243
244        // Make sure we add a paid song before any "free" songs.
245        int iPaidQueuedCnt = 0;
246        boolean bDone = false;
247
248        if (playListVector.size() > 0)
249        {
250           for (int i = 0; (i < playListVector.size() - 1) && !bDone; i++)
251           {
252              iPaidQueuedCnt = ((PlayListEntry)playListVector.elementAt(i)).getPaidQueuedCnt();
253
254              if (iPaidQueuedCnt == 0)
255              {
256                 playListVector.insertElementAt(mp3, i);
257
258                 bDone = true;
259              }
260           }
261        }
262
263        // There weren't any free songs in the queue, therefore (using FIFO), insert at the end.
264        if (bDone == false)
265        {
266           playListVector.addElement(mp3);
267        }
268     }
269
270     public void addToPlayList(PlayListEntry mp3)
271     {
```

```
272         playListVector.addElement(mp3);
273         mp3.incrementQueuedCnt();
274      }
275
276      public void addToPlayList(PlayListEntry mp3, int iPos)
277      {
278         playListVector.insertElementAt(mp3, iPos);
279         mp3.incrementQueuedCnt();
280      }
281
282      public void addToPlayList(Vector vector)
283      {
284         Object userObject;
285
286         for (Enumeration e = vector.elements(); e.hasMoreElements(); )
287         {
288            userObject = e.nextElement();
289
290            if (userObject instanceof PlayListEntry)
291            {
292               playListVector.addElement(((PlayListEntry)userObject));
293
294               ((PlayListEntry)userObject).incrementQueuedCnt();
295            }
296         }
297      }
298
299      public void removeFromPlayList(Vector vector)
300      {
301         Object userObject;
302
303         for (Enumeration e = vector.elements(); e.hasMoreElements(); )
304         {
305            userObject = e.nextElement();
306
307            if (userObject instanceof PlayListEntry)
308            {
309               PlayListEntry mp3 = ((PlayListEntry)userObject);
310
311               playListVector.removeElement(mp3);
312
313               if (mp3.getPaidQueuedCnt() > 0)
314               {
315                  mp3.decrementPaidQueuedCnt();
316                  mp3.incrementPaidCnt();
317               }
318
319               if (mp3.getQueuedCnt() > 0)
320               {
321                  mp3.decrementQueuedCnt();
322                  mp3.incrementPlayedCnt();
323               }
324            }
325         }
326      }
327
328      public boolean removeFromPlayList(PlayListEntry mp3)
329      {
330         boolean bDone = false;
331
332         for (int i = 0; (i < playListVector.capacity()) && !bDone; i++)
333         {
334            if (mp3.equals((PlayListEntry)playListVector.elementAt(i)))
335            {
336               playListVector.removeElementAt(i);
337
338               if (mp3.getPaidQueuedCnt() > 0)
339               {
340                  mp3.decrementPaidQueuedCnt();
341                  mp3.incrementPaidCnt();
342               }
343
344               if (mp3.getQueuedCnt() > 0)
345               {
346                  mp3.decrementQueuedCnt();
347                  mp3.incrementPlayedCnt();
348               }
349
350               bDone = true;
351            }
352         }
353
354         if (bDone)
355            return true;
356         else
357            return false;
358      }
359
360      public void flushPlayList()
361      {
```

```
362         for (Enumeration e = playListVector.elements(); e.hasMoreElements(); )
363         {
364            ((PlayListEntry)e.nextElement()).resetQueuedCnt();
365         }
366
367         playListVector.removeAllElements();
368      }
369
370      public String getTimeRemaining()
371      {
372         String  strTime = null;
373         String  strMin  = null;
374         String  strSec  = null;
375         Integer intSec  = null;
376         Integer intMin  = null;
377
378         if (iTimeRemaining < 3600)
379         {
380            intMin = new Integer(iTimeRemaining / 60);
381            intSec = new Integer(iTimeRemaining % 60);
382
383            // Convert the minutes portion.
384            if (intMin.intValue() == 0)
385               strMin = "00";
386            else
387               if ( (intMin.intValue() > 0) && (intMin.intValue() < 10) )
388               {
389                  strMin = "0";
390                  strMin = strMin.concat(intMin.toString());
391               }
392               else
393                  strMin = intMin.toString();
394
395            // Convert the seconds portion.
396            if (intSec.intValue() == 0)
397               strSec = "00";
398            else
399               if ( (intSec.intValue() > 0) && (intSec.intValue() < 10) )
400               {
401                  strSec = "0";
402                  strSec = strSec.concat(intSec.toString());
403               }
404               else
405                  strSec = intSec.toString();
406
407            // Now, put it all together.
408            strTime = strMin + ":" + strSec;
409         }
410         else
411         {
412            System.out.println("Time Remaining in seconds: " + iTimeRemaining);
413            strTime = new String( new Integer(iTimeRemaining).toString() );
414         }
415
416         strMin  = null;
417         strSec  = null;
418         intSec  = null;
419         intMin  = null;
420
421         return strTime;
422      }
423
424   }
```

ATTACHMENT 5

```
/**
 * Filename: WinAmpMgr.java
 *
 * Author: Tom Myers
 *
 * Version: 1.0
 *
 * Purpose: This file contains the code for the WinAmpPlayerMgr object class, which is a
 *          concrete sub-class of the PlayerMgr abstract class.  That is, this class has
 *          implmented all of the required methods for interacting, in this case, the
 *          Winamp MP3 player (a native, windows-based freeware software program)  This
 *          interaction with Winamp is accomplished via the use Java Native Interface methods
 *          that are wrappers around the appropriate functions found in Amp.DLL.  Amp.C and
 *          WinAmpMgr.h are related to the creation of Amp.DLL and are created by Tom Myers.
 *          Frontend.h however, is a header file supplied by Nullsoft, the makers of Winamp,
 *          and therefore, should not be a part of the patent application (whereas, Amp.DLL
 *          would be)
 *
 * Inputs: None
 *
 * Outputs: None
 *
 * Development Environment:  JDK 1.3 (Sun's Java 2 Standard Edition version 1.3) was used.
 *                           Microsoft Visual C++ 6.0 (to build Amp.DLL, which is a
 *                           required library for this class)
 *
 *
 * (c) Copyright 2000 Digital Jukebox Technologies LLC.  All Rights Reserved.
 */

import java.io.*;

import javax.swing.*;

import WinampFilter.*;
import CustomFileView.*;
import PlayerMgr.*;
import TreeMgr.*;

public class WinAmpMgr extends PlayerMgr
{
   public native int getWinAmpStatus();
   public native int playWinAmp(String strMp3);
   public native int clearWinAmpPlayList();
   public native int getWinAmpOutputTime();
   public native int getWinAmpSongLength();
   public native int closeWinAmp();
   public native int pressPlayWinAmp();
   public native int pressPauseWinAmp();
   public native int pressStopWinAmp();
   public native int setWinAmpVolume(int iVol);

   static
   {
      try
      {
         System.loadLibrary("Amp");
      }
      catch (Exception excptn)
      {
         System.out.println(excptn.toString());
      }
   }

   private String  _mp3Exec;
   private File    winAmpExe;
   private boolean bJustStartedPlaying;
   private int     iOutputTime;

   public WinAmpMgr()
   {
      super();

      _mp3Exec = "winamp.exe";

      winAmpExe = new File(_mp3Exec);

      if (!winAmpExe.exists())
      {
         _mp3Exec = "c:\\program files\\winamp\\winamp.exe";

         winAmpExe = new File(_mp3Exec);
         if (!winAmpExe.exists())
         {
            //JOptionPane pane   = new JOptionPane("Winamp was not found!  Please select the version of Winamp
you would like to use", JOptionPane.INFORMATION_MESSAGE);
            //JDialog infoDialog = pane.createDialog(this, "Information");
            //infoDialog.show();

            JFileChooser  chooser = new JFileChooser(".\\");
```

```
90                chooser.setFileFilter(new WinampFilter());
91                chooser.setSelectedFile(null);
92                //chooser.setFileView(new CustomFileView());
93
94                int state = chooser.showOpenDialog(null);
95                boolean bDone = false;
96
97                while (!bDone)
98                {
99                   if (state == JFileChooser.APPROVE_OPTION)
100                  {
101                     File file = chooser.getSelectedFile();
102
103                     if (file.toString().toLowerCase().indexOf("winamp.exe") != -1)
104                     {
105                        bDone = true;
106                        _mp3Exec  = file.toString();
107                        winAmpExe = file;
108                     }
109                  }
110               }
111            }
112         }
113         bJustStartedPlaying = false;
114         iOutputTime = 0;
115
116         startWinamp();
117      }
118
119      public void startWinamp()
120      {
121         // Start up Winamp.
122         try
123         {
124            Runtime.getRuntime().exec(_mp3Exec);
125         }
126         catch(java.io.IOException e)
127         {
128            System.out.println("Could not start Winamp!!!");
129         }
130
131         try
132         {
133            sleep(2000);
134         }
135         catch (Exception excptn)
136         {
137            System.out.println(excptn.toString());
138         }
139      }
140
141      public String getWinamp()
142      {
143         return _mp3Exec;
144      }
145
146      public void setWinamp(File file)
147      {
148         _mp3Exec  = file.toString();
149         winAmpExe = file;
150      }
151
152      public void setVolume(int iVol)
153      {
154         iVolume = iVol;
155         setWinAmpVolume((int)(iVol * 2.55));
156      }
157
158      public void cleanUp()
159      {
160         iStatus = STOPPED;
161
162         int iRes = pressStopWinAmp();
163
164         //iRes = closeWinAmp();
165      }
166
167      public int getStatus()
168      {
169         return getWinAmpStatus();
170      }
171
172      public int getOutputTime()
173      {
174         return getWinAmpOutputTime();
175      }
176
177      public int getSongLength()
178      {
179         return getWinAmpSongLength();
```

```
180      }
181
182      public void pressPause()
183      {
184         if (iStatus == PAUSED)
185         {
186            iStatus = PLAYING;
187            int iRes = pressPauseWinAmp();
188         }
189         else
190         {
191            iStatus = PAUSED;
192            int iRes = pressPauseWinAmp();
193         }
194         return;
195      }
196
197      public void pressStop()
198      {
199         iStatus = STOPPED;
200         int iRes = pressStopWinAmp();
201         currentPlayListObj = null;
202
203         return;
204      }
205
206      public void play(PlayListEntry mp3)
207      {
208         iStatus = PLAYING;
209
210         setLockOnQueue(true);
211         try
212         {
213            int iRes = clearWinAmpPlayList();
214            iRes = clearWinAmpPlayList();
215
216            Runtime.getRuntime().exec(_mp3Exec + " \"" +  mp3.getMp3Path() + "\"");
217
218            strCurrentSong = mp3.getMp3Path();
219            currentPlayListObj = mp3;
220
221            bJustStartedPlaying = true;
222         }
223         catch (Exception excptn)
224         {
225            System.out.println(excptn.toString());
226         }
227         setLockOnQueue(false);
228      }
229
230      public void run()
231      {
232         // Run forever.
233         while (true)
234         {
235            while (iStatus != STOPPED)
236            {
237               iStatus     = getWinAmpStatus();
238               iOutputTime = getWinAmpOutputTime();
239               iSongLength = getWinAmpSongLength();
240
241
242               // If Winamp can't run properly on the computer, then nothing we can do...
243               if (iStatus == -1)
244               {
245                  bLockOnQueue = true;;
246               }
247
248               if (bJustStartedPlaying == true)
249               {
250                  iStatus = PLAYING;
251                  bJustStartedPlaying = false;
252               }
253               else
254               {
255                  if ( (iOutputTime > 0) && (iSongLength > 0) )
256                     iTimeRemaining = ((iSongLength * 1000) - iOutputTime) / 1000;
257                  else
258                     iTimeRemaining = 0;
259               }
260
261               try
262               {
263                  sleep(1000);
264               }
265               catch (Exception excptn)
266               {
267                  System.out.println(excptn.toString());
268               }
269            }
```

```java
            if ( (!playListVector.isEmpty()) && (!bLockOnQueue) && (!bFirstTime) )
            {
               PlayListEntry mp3 = ((PlayListEntry)playListVector.firstElement());

               if (mp3.getPaidQueuedCnt() == 0)
               {
                  bIsCurrSongFree = true;
               }
               else
               {
                  bIsCurrSongFree = false;
               }

               play(mp3);

               removeFromPlayList(mp3);

               try
               {
                  sleep(2000);
               }
               catch (Exception excptn)
               {
                  System.out.println(excptn.toString());
               }
            }
            else
            {
               // Go to sleep, waking up one a sec. to see if there are any songs to play.
               try
               {
                  sleep(1000);
               }
               catch (Exception excptn)
               {
                  System.out.println(excptn.toString());
               }
            }
         }
      }
}
```

ATTACHMENT 6

```
/**
 * Filename: CDPanel.java
 *
 * Author: Tom Myers
 *
 * Version: 1.0
 *
 * Purpose: This file contains the code for the CDPanel object class, used to
 *          to display the cover art, track listings, artist, CD title, and genre type
 *          to the user.  Instances of this type are used in the main panel, search panel,
 *          popular panel and the genre panel.
 *
 * Inputs: The following parameters are used to construct objects of this class:
 *          1.iNewCDAgeThreshold - An integer that is used to determine whether or not a
 *          CDPanel is "new" or not.  A "new" CDPanel is displayed with a red border.
 *          2. bShowQ - A boolean that when true, will display the "Q" icon next to a track that
 *          is already in the song queue.
 *          3. iLev0 - An integer that determines the minimum number of plays needed for a song
 *          to have the "one green bar" icon displayed next to it.
 *          4. iLev1 - An integer that determines the minimum number of plays needed for a song
 *          to have the "two green bars" icon displayed next to it.
 *          5. iLev2 - An integer that determines the minimum number of plays needed for a song
 *          to have the "three green bars" icon displayed next to it.
 *          6. image - A 250x250 JPEG IconImage that holds a rendition of the CD cover art to be
 *          displayed.
 *          7. listData - A Vector of PlayListEntry objects containing the track listings for the
 *          CD.
 *          8. strCDTitle - A String that holds the CD Title.
 *          9. strGenre - A String that holds the genre type.
 *          10. selTxtFld - The JTextField from the main panel that holds the selection text.
 *          11. cnclBtn - The JButton from the main panel that corresponds to the Cancel button.
 *
 *
 * Outputs: None
 *
 * Development Environment:  JDK 1.3 (Sun's Java 2 Standard Edition version 1.3) was used.
 *
 *
 * (c) Copyright 2000 Digital Jukebox Technologies LLC.  All Rights Reserved.
 */

import java.util.Vector;
import java.awt.*;
import java.awt.event.MouseAdapter;
import javax.swing.*;
import javax.swing.border.LineBorder;
import javax.swing.event.ListSelectionListener;

import TreeMgr.*;
import MyListRenderer.*;

public class CDPanel extends JPanel
{
    private JLabel      instruct;
    private JLabel      cover;
    private JScrollPane scrollpane;
    private JList       listbox;
    private Vector      listboxvect;
    private JTextField  txtfield;
    private JTextField  genreTextField;
    private JTextField  selectionTxtField = null;
    private JButton     cancelBtn = null;

    public CDPanel(int iNewCDAgeThreshold, boolean bShowQ, int iLev0, int iLev1, int iLev2, ImageIcon image,
Vector listData, String strCDTitle, String strGenre)
    {
        this(iNewCDAgeThreshold, bShowQ, iLev0, iLev1, iLev2, image, listData, strCDTitle, strGenre, null, null);
    }

    public CDPanel(int iNewCDAgeThreshold, boolean bShowQ, int iLev0, int iLev1, int iLev2, ImageIcon image,
Vector listData, String strCDTitle, String strGenre, JTextField selTxtFld, JButton cnclBtn)
    {
        super();
        setSize(509,295);
        setLayout(null);
        setForeground(Color.white);
        setBackground(Color.black);

        selectionTxtField = selTxtFld;
        cancelBtn = cnclBtn;
        listboxvect = listData;

        instruct = new JLabel(new ImageIcon("images/instruct.gif"));
        instruct.setBounds(1,21,250,25);
        instruct.setForeground(Color.white);
        instruct.setBackground(Color.black);

        txtfield = new JTextField(strCDTitle);
        txtfield.setHorizontalAlignment(JTextField.LEFT);
        txtfield.setFont(new Font("SansSerif", Font.BOLD, 12));
```

```
89          txtfield.setBounds(1,1,383,20);
90          txtfield.setEditable(false);
91          txtfield.setAutoscrolls(false);
92          txtfield.setScrollOffset(0);
93          txtfield.setCaretPosition(0);
94          txtfield.setBorder(new LineBorder(Color.black, 1));
95
96          genreTextField = new JTextField(strGenre + "  ");
97          genreTextField.setHorizontalAlignment(JTextField.RIGHT);
98          genreTextField.setFont(new Font("SansSerif", Font.BOLD, 12));
99          genreTextField.setBounds(383,1,125,20);
100         genreTextField.setEditable(false);
101         genreTextField.setAutoscrolls(false);
102         genreTextField.setBorder(new LineBorder(Color.black, 1));
103
104         cover = new JLabel(image);
105         cover.setBounds(1,45,250,250);
106         cover.setForeground(Color.white);
107         cover.setBackground(Color.black);
108         if (selectionTxtField != null)
109         {
110            cover.addMouseListener(
111               new MouseAdapter()
112               {
113                  public void mouseClicked(java.awt.event.MouseEvent event)
114                  {
115                     selectionTxtField.setText(txtfield.getText().substring(0,3));
116                     cancelBtn.setEnabled(true);
117                  }
118               });
119         }
120
121         listbox = new JList(listboxvect);
122         listbox.setCellRenderer(new MyListRenderer(iLev0, iLev1, iLev2, bShowQ));
123         listbox.setForeground(Color.white);
124         listbox.setBackground(Color.black);
125         listbox.setBounds(0,0,257,290);
126         listbox.addListSelectionListener(
127            new ListSelectionListener()
128            {
129               public void valueChanged(javax.swing.event.ListSelectionEvent event)
130               {
131                  int i = listbox.getMaxSelectionIndex();
132
133                  if (i >= 0 && i < listboxvect.size())
134                  {
135                     if (listboxvect.elementAt(i) instanceof PlayListEntry)
136                     {
137                        PlayListEntry mp3         = (PlayListEntry)listboxvect.elementAt(i);
138                        String        strTrackNum = null;
139                        int           iTrackNum   = mp3.getTrackNum();
140                        String        selection   = null;
141
142                        if (iTrackNum < 10)
143                           strTrackNum = "0" + Integer.toString(iTrackNum);
144                        else
145                           strTrackNum = Integer.toString(iTrackNum);
146
147
148                        if (selectionTxtField != null)
149                        {
150                           // Now, update the selection text field.
151                           if (strTrackNum != null)
152                           {
153                              selection = txtfield.getText().substring(0,3) + strTrackNum;
154                              selectionTxtField.setText(selection);
155                              cancelBtn.setEnabled(true);
156                           }
157                        }
158                     }
159                  }
160               }
161            });
162
163         scrollpane = new JScrollPane();
164         scrollpane.setBounds(251,21,257,273);
165
166         scrollpane.setForeground(Color.white);
167         scrollpane.setBackground(Color.black);
168         scrollpane.getViewport().add(listbox);
169
170         JScrollBar horizontal = scrollpane.getHorizontalScrollBar();
171         horizontal.setPreferredSize(new Dimension(horizontal.getWidth(),25));
172
173         JScrollBar vertical = scrollpane.getVerticalScrollBar();
174         vertical.setPreferredSize(new Dimension(25,vertical.getHeight()));
175
176         setColorByAge(iNewCDAgeThreshold);
177
178         add(txtfield);
```

```
179         add(genreTextField);
180         add(instruct);
181         add(cover);
182         add(scrollpane);
183      }
184
185      public void init(ImageIcon image, Vector listData, String strCDTitle, String strGenre)
186      {
187         cover.setIcon(image);
188         listbox.setListData(listData);
189         listbox.repaint(listbox.getVisibleRect());
190         genreTextField.setText(strGenre);
191         txtfield.setText(strCDTitle);
192      }
193
194      public void die()
195      {
196         remove(txtfield);
197         remove(genreTextField);
198         remove(instruct);
199         remove(cover);
200         remove(scrollpane);
201
202         instruct = null;
203         cover = null;
204         scrollpane = null;
205         listbox = null;
206         listboxvect = null;
207         txtfield = null;
208         genreTextField = null;
209      }
210
211      public void forceRepaint()
212      {
213         listbox.repaint(listbox.getVisibleRect());
214      }
215
216      public String getCDNumber()
217      {
218         String strCDNumber = "";
219
220         if (txtfield.getText().length() >= 3)
221            strCDNumber = txtfield.getText().substring(0,3);
222
223         return strCDNumber;
224      }
225
226      public void clearSelection()
227      {
228         listbox.clearSelection();
229      }
230
231      public void setSelectedSong(PlayListEntry mp3)
232      {
233         PlayListEntry tmpMp3 = null;
234         boolean bDone = false;
235         for (int i = 0; i < listboxvect.size() && !bDone; i++)
236         {
237            if (listboxvect.elementAt(i) instanceof PlayListEntry)
238            {
239               tmpMp3 = (PlayListEntry)listboxvect.elementAt(i);
240               if (tmpMp3.getMp3Path().equalsIgnoreCase(mp3.getMp3Path()))
241               {
242                  bDone = true;
243                  listbox.setSelectedIndex(i);
244               }
245            }
246         }
247      }
248
249
250      /** Sets the color of the border of the CDPanel based upon the age of the first mp3 in the list.
251       *  The ages is an integer value that represents the number of elapsed days since 1/1/2000.
252       */
253      private void setColorByAge(int iNewCDAgeThreshold)
254      {
255         PlayListEntry firstMp3 = (PlayListEntry)listboxvect.elementAt(0);
256         int iAge = firstMp3.getAge();
257
258         if (iAge <= iNewCDAgeThreshold)
259         {
260            setBorder(new LineBorder(Color.red, 1));
261            instruct.setBorder(new LineBorder(Color.red, 1));
262            cover.setBorder(new LineBorder(Color.red, 1));
263            scrollpane.setBorder(new LineBorder(Color.red, 1));
264
265            txtfield.setForeground(Color.white);
266            txtfield.setBackground(Color.red);
267
268            genreTextField.setForeground(Color.white);
```

```
269             genreTextField.setBackground(Color.red);
270          }
271          else
272          {
273             setBorder(new LineBorder(Color.white, 1));
274             instruct.setBorder(new LineBorder(Color.white, 1));
275             cover.setBorder(new LineBorder(Color.white, 1));
276             scrollpane.setBorder(new LineBorder(Color.white, 1));
277
278             txtfield.setForeground(Color.black);
279             txtfield.setBackground(Color.white);
280
281             genreTextField.setForeground(Color.black);
282             genreTextField.setBackground(Color.white);
283          }
284       }
285    }
```

ATTACHMENT 7

```
/**
 * Filename: KeyboardPanel.java
 *
 * Author: Tom Myers
 *
 * Version: 1.0
 *
 * Purpose: Used to provide the "soft-keyboard" functionality used for the Search Panel and
 *          Logon Panel screens.
 *
 * Inputs:
 *
 * Outputs:
 *
 * Development Environment:  JDK 1.3 (Sun's Java 2 Standard Edition version 1.3) was used.
 *
 *
 * (c) Copyright 2000 Digital Jukebox Technologies LLC.  All Rights Reserved.
 */


import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.Vector;

import javax.swing.*;
import javax.swing.ImageIcon;
import javax.swing.Icon;
import javax.swing.Timer;
import javax.swing.border.LineBorder;
import javax.swing.event.ListSelectionListener;


public class KeyboardPanel extends JPanel implements ActionListener
{
    private JTextField     textField1 = null;
    private JPasswordField textField2 = null;

    private JTextField textField  = null;

    private String      userid;
    private String      passwd;

    private boolean     bIsChgPw = false;
    private String      newpasswd;

    private JButton     btn_Logon;
    private JButton     btn_ChangePassword;


    private boolean     bShiftLock  = false;
    private boolean     bCapsLock   = false;
    private boolean     bInsertMode = false;


    private JButton     btn_esc = null;

    private JButton     btn_f1  = null;
    private JButton     btn_f2  = null;
    private JButton     btn_f3  = null;
    private JButton     btn_f4  = null;

    private JButton     btn_f5  = null;
    private JButton     btn_f6  = null;
    private JButton     btn_f7  = null;
    private JButton     btn_f8  = null;

    private JButton     btn_f9  = null;
    private JButton     btn_f10 = null;
    private JButton     btn_f11 = null;
    private JButton     btn_f12 = null;

    private JButton     btn_PrintScreen = null;
    private JButton     btn_ScrollLock  = null;
    private JButton     btn_Pause       = null;

    private JButton     btn_LeftQuote = null;
    private JButton     btn_1  = null;
    private JButton     btn_2  = null;
    private JButton     btn_3  = null;
    private JButton     btn_4  = null;
    private JButton     btn_5  = null;
    private JButton     btn_6  = null;
    private JButton     btn_7  = null;
    private JButton     btn_8  = null;
    private JButton     btn_9  = null;
    private JButton     btn_0  = null;
    private JButton     btn_Minus  = null;
    private JButton     btn_Equals = null;
```

```
91      private JButton     btn_BackSpace  = null;
92
93      private JButton     btn_Tab  = null;
94      private JButton     btn_q  = null;
95      private JButton     btn_w  = null;
96      private JButton     btn_e  = null;
97      private JButton     btn_r  = null;
98      private JButton     btn_t  = null;
99      private JButton     btn_y  = null;
100     private JButton     btn_u  = null;
101     private JButton     btn_i  = null;
102     private JButton     btn_o  = null;
103     private JButton     btn_p  = null;
104     private JButton     btn_LeftBracket  = null;
105     private JButton     btn_RightBracket = null;
106     private JButton     btn_BackSlash    = null;
107
108     private JButton     btn_CapsLock = null;
109     private JButton     btn_a  = null;
110     private JButton     btn_s  = null;
111     private JButton     btn_d  = null;
112     private JButton     btn_f  = null;
113     private JButton     btn_g  = null;
114     private JButton     btn_h  = null;
115     private JButton     btn_j  = null;
116     private JButton     btn_k  = null;
117     private JButton     btn_l  = null;
118     private JButton     btn_SemiColon = null;
119     private JButton     btn_Quote  = null;
120     private JButton     btn_Enter  = null;
121
122     private JButton     btn_Shift  = null;
123     private JButton     btn_z  = null;
124     private JButton     btn_x  = null;
125     private JButton     btn_c  = null;
126     private JButton     btn_v  = null;
127     private JButton     btn_b  = null;
128     private JButton     btn_n  = null;
129     private JButton     btn_m  = null;
130     private JButton     btn_Comma         = null;
131     private JButton     btn_Period        = null;
132     private JButton     btn_ForwardSlash  = null;
133     private JButton     btn_Shift2 = null;
134
135     private JButton     btn_Ctrl  = null;
136     private JButton     btn_Alt   = null;
137     private JButton     btn_Space = null;
138     private JButton     btn_Alt2  = null;
139     private JButton     btn_Ctrl2 = null;
140
141     private JButton     btn_Insert = null;
142     private JButton     btn_Home   = null;
143     private JButton     btn_PageUp = null;
144     private JButton     btn_Delete = null;
145     private JButton     btn_End    = null;
146     private JButton     btn_PageDn = null;
147
148     private JButton     btn_Up    = null;
149     private JButton     btn_Left  = null;
150     private JButton     btn_Down  = null;
151     private JButton     btn_Right = null;
152
153     public String getUserid()
154     {
155        return userid;
156     }
157
158     public String getPassword()
159     {
160        return passwd;
161     }
162
163     public String getChangePassword()
164     {
165        newpasswd = textField.getText();
166
167        return newpasswd;
168     }
169
170     public String getTextFieldText()
171     {
172        return textField.getText();
173     }
174
175     public void setChangePasswordTextField()
176     {
177        bIsChgPw   = true;
178        textField  = textField2;
179        textField.setText("");
180     }
```

```
182       public void clearPasswordField()
183       {
184          textField  = textField2;
185          textField.setText("");
186       }
187
188       public void setPasswordTextField()
189       {
190          bIsChgPw   = false;
191          textField  = textField2;
192          textField.setText("");
193       }
194
195       public KeyboardPanel(int x1, int y1, int x2, int y2, JTextField txtField1)
196       {
197          this(x1, y1, x2, y2, txtField1, null, "", "", "", null, null);
198       }
199
200       public KeyboardPanel(int x1, int y1, int x2, int y2, JTextField txtField1, JPasswordField txtField2, String
uid, String pwd, String newpwd, JButton btnLgn, JButton btnChg)
201       {
202          super();
203
204          setForeground(Color.white);
205          setBackground(Color.black);
206
207          setLayout(null);
208
209          setBounds(x1, y1, x2, y2);
210
211          textField1 = txtField1;
212          textField  = txtField1;
213
214          userid = uid;
215          passwd = pwd;
216          newpasswd = newpwd;
217
218          btn_Logon = btnLgn;
219          btn_ChangePassword = btnChg;
220
221
222          initializeKeyboardKeys();
223
224          if (txtField2 != null)
225          {
226             textField2 = txtField2;
227             btn_Tab.setEnabled(true);
228             btn_Enter.setEnabled(true);
229          }
230          else
231          {
232             textField2 = null;
233          }
234
235
236          setVisible(true);
237       }
238
239       public void actionPerformed(java.awt.event.ActionEvent e)
240       {
241          Object object = e.getSource();
242
243          {
244             StringBuffer strTxt = new StringBuffer(textField.getText());
245
246             int iCaretPos = textField.getCaretPosition();
247
248             if (object == btn_Shift || object == btn_Shift2)
249                if (!bShiftLock)
250                {
251                   setNonAlphaKeysToUpper();
252
253                   btn_Shift.setBackground(new Color(148,148,148));
254                   btn_Shift2.setBackground(new Color(148,148,148));
255                   bShiftLock = true;
256                }
257                else
258                {
259                   setNonAlphaKeysToLower();
260
261                   btn_Shift.setBackground(new Color(192,192,192));
262                   btn_Shift2.setBackground(new Color(192,192,192));
263                   bShiftLock = false;
264                }
265             else if (object == btn_CapsLock)
266                if (!bCapsLock)
267                {
268                   btn_CapsLock.setBackground(new Color(148,148,148));
269                   bCapsLock = true;
270                }
```

```
271             else
272             {
273                btn_CapsLock.setBackground(new Color(192,192,192));
274                bCapsLock = false;
275             }
276          else if (object == btn_LeftQuote)
277             if (!bShiftLock)
278                strTxt.insert(iCaretPos, '`');
279             else
280                strTxt.insert(iCaretPos, '~');
281          else if (object == btn_1)
282             if (!bShiftLock)
283                strTxt.insert(iCaretPos, '1');
284             else
285                strTxt.insert(iCaretPos, '!');
286          else if (object == btn_2)
287             if (!bShiftLock)
288                strTxt.insert(iCaretPos, '2');
289             else
290                strTxt.insert(iCaretPos, '@');
291          else if (object == btn_3)
292             if (!bShiftLock)
293                strTxt.insert(iCaretPos, '3');
294             else
295                strTxt.insert(iCaretPos, '#');
296          else if (object == btn_4)
297             if (!bShiftLock)
298                strTxt.insert(iCaretPos, '4');
299             else
300                strTxt.insert(iCaretPos, '$');
301          else if (object == btn_5)
302             if (!bShiftLock)
303                strTxt.insert(iCaretPos, '5');
304             else
305                strTxt.insert(iCaretPos, '%');
306          else if (object == btn_6)
307             if (!bShiftLock)
308                strTxt.insert(iCaretPos, '6');
309             else
310                strTxt.insert(iCaretPos, '^');
311          else if (object == btn_7)
312             if (!bShiftLock)
313                strTxt.insert(iCaretPos, '7');
314             else
315                strTxt.insert(iCaretPos, '&');
316          else if (object == btn_8)
317             if (!bShiftLock)
318                strTxt.insert(iCaretPos, '8');
319             else
320                strTxt.insert(iCaretPos, '*');
321          else if (object == btn_9)
322             if (!bShiftLock)
323                strTxt.insert(iCaretPos, '9');
324             else
325                strTxt.insert(iCaretPos, '(');
326          else if (object == btn_0)
327             if (!bShiftLock)
328                strTxt.insert(iCaretPos, '0');
329             else
330                strTxt.insert(iCaretPos, ')');
331          else if (object == btn_Minus)
332             if (!bShiftLock)
333                strTxt.insert(iCaretPos, '-');
334             else
335                strTxt.insert(iCaretPos, '-');
336          else if (object == btn_Equals)
337             if (!bShiftLock)
338                strTxt.insert(iCaretPos, '=');
339             else
340                strTxt.insert(iCaretPos, '+');
341          else if (object == btn_BackSpace)
342             if (iCaretPos > 0)
343             {
344                iCaretPos = iCaretPos - 1;
345                strTxt.deleteCharAt(iCaretPos);
346                textField.setCaretPosition(iCaretPos);
347             }
348             else
349                iCaretPos = 0;
350          else if (object == btn_Tab || object == btn_Enter)
351          {
352             if (textField == textField1)
353             {
354                userid = textField.getText();
355
356                textField = (JTextField)textField2;
357                textField1.setText(userid);
358
359                textField.setText("");
360                textField2.setText("");
```

```
            }
            else
            {
               if (bIsChgPw == false)
               {
                  passwd = textField.getText();
                  textField = textField1;
               }
               else
               {
                  newpasswd = textField.getText();
               }
            }

            strTxt = new StringBuffer("");
         }
         else if (object == btn_q)
            if (bShiftLock ^ bCapsLock)
               strTxt.insert(iCaretPos, 'Q');
            else
               strTxt.insert(iCaretPos, 'q');
         else if (object == btn_w)
            if (bShiftLock ^ bCapsLock)
               strTxt.insert(iCaretPos, 'W');
            else
               strTxt.insert(iCaretPos, 'w');
         else if (object == btn_e)
            if (bShiftLock ^ bCapsLock)
               strTxt.insert(iCaretPos, 'E');
            else
               strTxt.insert(iCaretPos, 'e');
         else if (object == btn_r)
            if (bShiftLock ^ bCapsLock)
               strTxt.insert(iCaretPos, 'R');
            else
               strTxt.insert(iCaretPos, 'r');
         else if (object == btn_t)
            if (bShiftLock ^ bCapsLock)
               strTxt.insert(iCaretPos, 'T');
            else
               strTxt.insert(iCaretPos, 't');
         else if (object == btn_y)
            if (bShiftLock ^ bCapsLock)
               strTxt.insert(iCaretPos, 'Y');
            else
               strTxt.insert(iCaretPos, 'y');
         else if (object == btn_u)
            if (bShiftLock ^ bCapsLock)
               strTxt.insert(iCaretPos, 'U');
            else
               strTxt.insert(iCaretPos, 'u');
         else if (object == btn_i)
            if (bShiftLock ^ bCapsLock)
               strTxt.insert(iCaretPos, 'I');
            else
               strTxt.insert(iCaretPos, 'i');
         else if (object == btn_o)
            if (bShiftLock ^ bCapsLock)
               strTxt.insert(iCaretPos, 'O');
            else
               strTxt.insert(iCaretPos, 'o');
         else if (object == btn_p)
            if (bShiftLock ^ bCapsLock)
               strTxt.insert(iCaretPos, 'P');
            else
               strTxt.insert(iCaretPos, 'p');
         else if (object == btn_LeftBracket)
            if (!bShiftLock)
               strTxt.insert(iCaretPos, '[');
            else
               strTxt.insert(iCaretPos, '{');
         else if (object == btn_RightBracket)
            if (!bShiftLock)
               strTxt.insert(iCaretPos, ']');
            else
               strTxt.insert(iCaretPos, '}');
         else if (object == btn_BackSlash)
            if (!bShiftLock)
               strTxt.insert(iCaretPos, '\\');
            else
               strTxt.insert(iCaretPos, '|');
         else if (object == btn_a)
            if (bShiftLock ^ bCapsLock)
               strTxt.insert(iCaretPos, 'A');
            else
               strTxt.insert(iCaretPos, 'a');
         else if (object == btn_s)
            if (bShiftLock ^ bCapsLock)
               strTxt.insert(iCaretPos, 'S');
            else
```

```
451                 strTxt.insert(iCaretPos, 's');
452              else if (object == btn_d)
453                 if (bShiftLock ^ bCapsLock)
454                    strTxt.insert(iCaretPos, 'D');
455                 else
456                    strTxt.insert(iCaretPos, 'd');
457              else if (object == btn_f)
458                 if (bShiftLock ^ bCapsLock)
459                    strTxt.insert(iCaretPos, 'F');
460                 else
461                    strTxt.insert(iCaretPos, 'f');
462              else if (object == btn_g)
463                 if (bShiftLock ^ bCapsLock)
464                    strTxt.insert(iCaretPos, 'G');
465                 else
466                    strTxt.insert(iCaretPos, 'g');
467              else if (object == btn_h)
468                 if (bShiftLock ^ bCapsLock)
469                    strTxt.insert(iCaretPos, 'H');
470                 else
471                    strTxt.insert(iCaretPos, 'h');
472              else if (object == btn_j)
473                 if (bShiftLock ^ bCapsLock)
474                    strTxt.insert(iCaretPos, 'J');
475                 else
476                    strTxt.insert(iCaretPos, 'j');
477              else if (object == btn_k)
478                 if (bShiftLock ^ bCapsLock)
479                    strTxt.insert(iCaretPos, 'K');
480                 else
481                    strTxt.insert(iCaretPos, 'k');
482              else if (object == btn_l)
483                 if (bShiftLock ^ bCapsLock)
484                    strTxt.insert(iCaretPos, 'L');
485                 else
486                    strTxt.insert(iCaretPos, 'l');
487              else if (object == btn_SemiColon)
488                 if (!bShiftLock)
489                    strTxt.insert(iCaretPos, ';');
490                 else
491                    strTxt.insert(iCaretPos, ':');
492              else if (object == btn_Quote)
493                 if (!bShiftLock)
494                    strTxt.insert(iCaretPos, "'");
495                 else
496                    strTxt.insert(iCaretPos, '"');
497              else if (object == btn_z)
498                 if (bShiftLock ^ bCapsLock)
499                    strTxt.insert(iCaretPos, 'Z');
500                 else
501                    strTxt.insert(iCaretPos, 'z');
502              else if (object == btn_x)
503                 if (bShiftLock ^ bCapsLock)
504                    strTxt.insert(iCaretPos, 'X');
505                 else
506                    strTxt.insert(iCaretPos, 'x');
507              else if (object == btn_c)
508                 if (bShiftLock ^ bCapsLock)
509                    strTxt.insert(iCaretPos, 'C');
510                 else
511                    strTxt.insert(iCaretPos, 'c');
512              else if (object == btn_v)
513                 if (bShiftLock ^ bCapsLock)
514                    strTxt.insert(iCaretPos, 'V');
515                 else
516                    strTxt.insert(iCaretPos, 'v');
517              else if (object == btn_b)
518                 if (bShiftLock ^ bCapsLock)
519                    strTxt.insert(iCaretPos, 'B');
520                 else
521                    strTxt.insert(iCaretPos, 'b');
522              else if (object == btn_n)
523                 if (bShiftLock ^ bCapsLock)
524                    strTxt.insert(iCaretPos, 'N');
525                 else
526                    strTxt.insert(iCaretPos, 'n');
527              else if (object == btn_m)
528                 if (bShiftLock ^ bCapsLock)
529                    strTxt.insert(iCaretPos, 'M');
530                 else
531                    strTxt.insert(iCaretPos, 'm');
532              else if (object == btn_Comma)
533                 if (!bShiftLock)
534                    strTxt.insert(iCaretPos, ',');
535                 else
536                    strTxt.insert(iCaretPos, '<');
537              else if (object == btn_Period)
538                 if (!bShiftLock)
539                    strTxt.insert(iCaretPos, '.');
540                 else
```

```
                strTxt.insert(iCaretPos, '>');
            else if (object == btn_ForwardSlash)
               if (!bShiftLock)
                  strTxt.insert(iCaretPos, '/');
               else
                  strTxt.insert(iCaretPos, '?');
            else if (object == btn_Space)
               strTxt.insert(iCaretPos, ' ');


            if (textField2 != null)
            {
               String strUpperCase = strTxt.toString().toUpperCase();

               textField.setText(strUpperCase);
            }
            else
            {
               textField.setText(strTxt.toString());
            }


            // Enable the Logon and Change Password buttons if we have the proper password length.
            if (btn_Logon != null)
            {
               if (strTxt.toString().length() >= 8)
               {
                  btn_Logon.setEnabled(true);
                  btn_ChangePassword.setEnabled(true);
               }
               else
               {
                  btn_Logon.setEnabled(false);
                  btn_ChangePassword.setEnabled(false);
               }
            }


            // Automatically advance to the Password field if we are dealing with the UserID.
            if (textField2 != null)
            {
               if (strTxt.toString().length() == 5 && textField == textField1)
               {
                  userid = textField.getText();

                  textField = (JTextField)textField2;
                  textField1.setText(userid);

                  textField.setText("");
                  textField2.setText("");

                  strTxt = new StringBuffer("");
               }
            }
         }
      }

      private void setNonAlphaKeysToUpper()
      {
         btn_LeftQuote.setText("~");
         btn_1.setText("!");
         btn_2.setText("@");
         btn_3.setText("#");
         btn_4.setText("$");
         btn_5.setText("%");
         btn_6.setText("^");
         btn_7.setText("&");
         btn_8.setText("*");
         btn_9.setText("(");
         btn_0.setText(")");
         btn_Minus.setText("_");
         btn_Equals.setText("+");
         btn_LeftBracket.setText("{");
         btn_RightBracket.setText("}");
         btn_BackSlash.setText("|");
         btn_SemiColon.setText(":");
         btn_Quote.setText( new Character('"').toString());
         btn_Comma.setText("<");
         btn_Period.setText(">");
         btn_ForwardSlash.setText("?");
      }

      private void setNonAlphaKeysToLower()
      {
         btn_LeftQuote.setText("`");
         btn_1.setText("1");
         btn_2.setText("2");
         btn_3.setText("3");
         btn_4.setText("4");
         btn_5.setText("5");
```

```
631         btn_6.setText("6");
632         btn_7.setText("7");
633         btn_8.setText("8");
634         btn_9.setText("9");
635         btn_0.setText("0");
636         btn_Minus.setText("-");
637         btn_Equals.setText("=");
638         btn_LeftBracket.setText("[");
639         btn_RightBracket.setText("]");
640         btn_BackSlash.setText("\\");
641         btn_SemiColon.setText(new Character(';').toString());
642         btn_Quote.setText("'");
643         btn_Comma.setText(",");
644         btn_Period.setText(".");
645         btn_ForwardSlash.setText("/");
646      }
647
648      private void initializeKeyboardKeys()
649      {
650         /*
651         btn_esc = new JButton("Esc"); initControl(btn_esc, 25,100,48,48, true);
652
653         btn_f1  = new JButton("F1"); initControl(btn_f1, 120,100,48,48, false);
654         btn_f2  = new JButton("F2"); initControl(btn_f2, 170,100,48,48, false);
655         btn_f3  = new JButton("F3"); initControl(btn_f3, 220,100,48,48, false);
656         btn_f4  = new JButton("F4"); initControl(btn_f4, 270,100,48,48, false);
657
658         btn_f5  = new JButton("F5"); initControl(btn_f5, 345,100,48,48, false);
659         btn_f6  = new JButton("F6"); initControl(btn_f6, 395,100,48,48, false);
660         btn_f7  = new JButton("F7"); initControl(btn_f7, 445,100,48,48, false);
661         btn_f8  = new JButton("F8"); initControl(btn_f8, 495,100,48,48, false);
662
663         btn_f9  = new JButton("F9"); initControl(btn_f9, 570,100,48,48, false);
664         btn_f10 = new JButton("F10"); initControl(btn_f10, 620,100,48,48, false);
665         btn_f11 = new JButton("F11"); initControl(btn_f11, 670,100,48,48, false);
666         btn_f12 = new JButton("F12"); initControl(btn_f12, 720,100,48,48, false);
667
668         btn_PrintScreen = new JButton("PrtScrn"); initControl(btn_PrintScreen, 800,100,48,48, false);
669         btn_ScrollLock  = new JButton("Scroll"); initControl(btn_ScrollLock, 850,100,48,48, false);
670         btn_Pause       = new JButton("Pause"); initControl(btn_Pause, 900,100,48,48, false);
671         */
672
673
674
675         btn_LeftQuote  = new JButton("`"); initControl(btn_LeftQuote, 25+45, 5,58,58, true);
676         btn_1  = new JButton("1"); initControl(btn_1,  85+45, 5,58,58, true);
677         btn_2  = new JButton("2"); initControl(btn_2, 145+45, 5,58,58, true);
678         btn_3  = new JButton("3"); initControl(btn_3, 205+45, 5,58,58, true);
679         btn_4  = new JButton("4"); initControl(btn_4, 265+45, 5,58,58, true);
680         btn_5  = new JButton("5"); initControl(btn_5, 325+45, 5,58,58, true);
681         btn_6  = new JButton("6"); initControl(btn_6, 385+45, 5,58,58, true);
682         btn_7  = new JButton("7"); initControl(btn_7, 445+45, 5,58,58, true);
683         btn_8  = new JButton("8"); initControl(btn_8, 505+45, 5,58,58, true);
684         btn_9  = new JButton("9"); initControl(btn_9, 565+45, 5,58,58, true);
685         btn_0  = new JButton("0"); initControl(btn_0, 625+45, 5,58,58, true);
686         btn_Minus  = new JButton("-"); initControl(btn_Minus, 685+45, 5,58,58, true);
687         btn_Equals = new JButton("="); initControl(btn_Equals, 745+45, 5,58,58, true);
688         btn_BackSpace  = new JButton("Back"); initControl(btn_BackSpace, 805+45, 5,105,58, true);
689
690         /*
691         btn_Insert = new JButton("Ins"); initControl(btn_Insert,  800,190,48,48, false);
692         btn_Home   = new JButton("Home"); initControl(btn_Home,   850,190,48,48, false);
693         btn_PageUp = new JButton("PgUp"); initControl(btn_PageUp, 900,190,48,48, false);
694         */
695
696
697
698         btn_Tab  = new JButton("Tab"); initControl(btn_Tab, 25+45,65,78,58, false);
699         btn_q  = new JButton("Q"); initControl(btn_q, 105+45,65,58,58, true);
700         btn_w  = new JButton("W"); initControl(btn_w, 165+45,65,58,58, true);
701         btn_e  = new JButton("E"); initControl(btn_e, 225+45,65,58,58, true);
702         btn_r  = new JButton("R"); initControl(btn_r, 285+45,65,58,58, true);
703         btn_t  = new JButton("T"); initControl(btn_t, 345+45,65,58,58, true);
704         btn_y  = new JButton("Y"); initControl(btn_y, 405+45,65,58,58, true);
705         btn_u  = new JButton("U"); initControl(btn_u, 465+45,65,58,58, true);
706         btn_i  = new JButton("I"); initControl(btn_i, 525+45,65,58,58, true);
707         btn_o  = new JButton("O"); initControl(btn_o, 585+45,65,58,58, true);
708         btn_p  = new JButton("P"); initControl(btn_p, 645+45,65,58,58, true);
709         btn_LeftBracket  = new JButton("["); initControl(btn_LeftBracket, 705+45,65,58,58, true);
710         btn_RightBracket = new JButton("]"); initControl(btn_RightBracket, 765+45,65,58,58, true);
711         btn_BackSlash    = new JButton("\\"); initControl(btn_BackSlash, 825+45,65,85,58, true);
712
713         /*
714         btn_Delete = new JButton("Del"); initControl(btn_Delete,  800,240,48,48, false);
715         btn_End    = new JButton("End"); initControl(btn_End,     850,240,48,48, false);
716         btn_PageDn = new JButton("PgDn"); initControl(btn_PageDn, 900,240,48,48, false);
717         */
718
719
720
```

```
721         btn_CapsLock = new JButton("Caps"); initControl(btn_CapsLock, 25+45,125,103,58, true);
722         btn_a  = new JButton("A"); initControl(btn_a, 130+45,125,58,58, true);
723         btn_s  = new JButton("S"); initControl(btn_s, 190+45,125,58,58, true);
724         btn_d  = new JButton("D"); initControl(btn_d, 250+45,125,58,58, true);
725         btn_f  = new JButton("F"); initControl(btn_f, 310+45,125,58,58, true);
726         btn_g  = new JButton("G"); initControl(btn_g, 370+45,125,58,58, true);
727         btn_h  = new JButton("H"); initControl(btn_h, 430+45,125,58,58, true);
728         btn_j  = new JButton("J"); initControl(btn_j, 490+45,125,58,58, true);
729         btn_k  = new JButton("K"); initControl(btn_k, 550+45,125,58,58, true);
730         btn_l  = new JButton("L"); initControl(btn_l, 610+45,125,58,58, true);
731         btn_SemiColon = new JButton(";"); initControl(btn_SemiColon, 670+45,125,58,58, true);
732         btn_Quote  = new JButton("'"); initControl(btn_Quote, 730+45,125,58,58, true);
733         btn_Enter  = new JButton("Enter"); initControl(btn_Enter, 790+45,125,120,58, false);
734
735
736
737
738         btn_Shift  = new JButton("Shift"); initControl(btn_Shift, 25+45,185,140,58, true);
739         btn_z  = new JButton("Z"); initControl(btn_z, 167+45,185,58,58, true);
740         btn_x  = new JButton("X"); initControl(btn_x, 227+45,185,58,58, true);
741         btn_c  = new JButton("C"); initControl(btn_c, 287+45,185,58,58, true);
742         btn_v  = new JButton("V"); initControl(btn_v, 347+45,185,58,58, true);
743         btn_b  = new JButton("B"); initControl(btn_b, 407+45,185,58,58, true);
744         btn_n  = new JButton("N"); initControl(btn_n, 467+45,185,58,58, true);
745         btn_m  = new JButton("M"); initControl(btn_m, 527+45,185,58,58, true);
746         btn_Comma         = new JButton(","); initControl(btn_Comma,        587+45,185,58,58, true);
747         btn_Period        = new JButton("."); initControl(btn_Period,       647+45,185,58,58, true);
748         btn_ForwardSlash  = new JButton("/"); initControl(btn_ForwardSlash, 707+45,185,58,58, true);
749         btn_Shift2 = new JButton("Shift"); initControl(btn_Shift2,          767+45,185,143,58, true);
750
751         /*
752         btn_Up    = new JButton("Up"); initControl(btn_Up, 850,340,48,48, true);
753         */
754
755
756         /*
757         btn_Ctrl  = new JButton("Ctrl"); initControl(btn_Ctrl,  25,260,100,58, false);
758         btn_Alt   = new JButton("Alt"); initControl(btn_Alt,   180,260,78,58, false);
759         */
760         btn_Space = new JButton(" "); initControl(btn_Space,   260+45,245,418,53, true);
761         /*
762         btn_Alt2  = new JButton("Alt"); initControl(btn_Alt2,  680,260,78,58, false);
763         btn_Ctrl2 = new JButton("Ctrl"); initControl(btn_Ctrl2,810,260,100,58, false);
764         */
765
766
767         /*
768         btn_Left  = new JButton("Left"); initControl(btn_Left,   800,260,58,58, true);
769         btn_Down  = new JButton("Down"); initControl(btn_Down,   850,260,58,58, true);
770         btn_Right = new JButton("Right"); initControl(btn_Right, 900,260,58,58, true);
771         */
772     }
773
774     private void initControl(JButton btn, int x1, int y1, int x2, int y2, boolean bEnable)
775     {
776         //btn.setBackground(Color.white);
777         btn.setForeground(Color.blue);
778         btn.setFont(new Font("SansSerif", Font.BOLD, 12));
779         btn.setBounds(x1,y1,x2,y2);
780         btn.setEnabled(bEnable);
781         btn.setFocusPainted(false);
782         btn.addActionListener(this);
783         add(btn);
784     }
785 }
```

ATTACHMENT 8

```
/**
 * Filename: SpinButton.java
 *
 * Author: Tom Myers
 *
 * Version: 1.0
 *
 * Purpose: To provide a custom screen control, ala OS/2's spinbutton GUI widget.
 *
 * Inputs:
 *
 * Outputs:
 *
 * Development Environment:  JDK 1.3 (Sun's Java 2 Standard Edition version 1.3) was used.
 *
 *
 * (c) Copyright 2000 Digital Jukebox Technologies LLC.  All Rights Reserved.
 */

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

/**
 * This class resembles the Spin Button GUI widget that we all know from the OS/2 platform.
 */
public class SpinButton extends JPanel implements ActionListener
{
   /** Holds the value that is displayed in the "entry field" portion of the spin button. */
   private int value;

   /** Holds the value that is displayed in the "entry field" portion of the spin button. */
   private Integer intValue;

   /** Holds the minimum value that the user can "spin" the value to. */
   private int minValue;

   /** Holds the maximum value that the user can "spin" the value to. */
   private int maxValue;

   /** Used for display of the value. */
   private JTextField valueTextField;

   /** Used to increment the value. */
   private JButton incrementButton;

   /** Used to decrement the value. */
   private JButton decrementButton;


   /**
    * Constructs a SpinButton object with a min. of 0, a max. of 100, and an initial value of 50.
    */
   public SpinButton()
   {
      // max, min, and initial values are the parameter order.
      this(100, 0, 50);
   }


   /**
    * Constructs a SpinButton object with a min. of 0, a max. of <code>iMax</code>,
    *  and an initial value of 0.
    * @param iMax The maximum value to use for the spin button entry field.  This field has
    * to be greater than zero.  The default is min=0, max=100, and initial value=50 otherwise.
    */
   public SpinButton(int iMax)
   {
      this(iMax, 0, 0);
   }


   /**
    * Constructs a SpinButton object with a min. of <code>iMin</code>, a max. of <code>iMax</code>,
    * and an initial value of <code>iMin</code>.  The minimum value must be less than the maximum
    * value.
    * @param iMax The maximum value to use for the spin button entry field.
    * @param iMin The minimum value to use for the spin button entry field.
    */
   public SpinButton(int iMax, int iMin)
   {
      this(iMax, iMin, iMin);
   }


   /**
    * Constructs a SpinButton object with a min. of <code>iMin</code>, a max. of <code>iMax</code>,
    *  and an initial value of <code>iInitialVal</code>.
    * @param iMax The maximum value to use for the spin button entry field.  This field has
```

```
91          * to be greater than zero.  The default is min=0, max=100, and initial value=50 otherwise.
92          * @param iMin The minimum value to use for the spin button entry field. This field has to be
93          * less than <code>iMax</code>.
94          * @param iInitialVal The initial value to use for the spin button entry field.  This field has
95          * to be greater than <code>iMin</code> yet less than <code>iMax</code>.
96          */
97         public SpinButton(int iMax, int iMin, int iInitialVal)
98         {
99            // Construct the JPanel object first.
100           super();
101           setLayout(null);
102           setSize(60,55);
103           setForeground(Color.white);
104           setBackground(Color.black);
105
106
107           // Make sure the arguments are valid.  Use the defaults if they aren't.
108           if (iMax <= 0 || iMin >= iMax || iInitialVal < iMin)
109           {
110              System.out.println("SpinButton::SpinButton():  Illegal argument(s) to constructor.");
111              System.out.println("SpinButton::SpinButton():  Using defaults of min=0,max=100 & val=50.");
112              iMin = 0;
113              iMax = 100;
114              iInitialVal = 50;
115           }
116
117           // Assign the data values for the passed in (or default) values.
118           minValue = iMin;
119           maxValue = iMax;
120           value = iInitialVal;
121           intValue = new Integer(value);
122
123
124           // Construct the screen elements and add them to the panel.
125           valueTextField = new JTextField(intValue.toString());
126           valueTextField.setBounds(2,18,30,20);
127           valueTextField.setEditable(false);
128           valueTextField.setFont(new Font("SansSerif", Font.BOLD, 12));
129           valueTextField.setForeground(Color.yellow);
130           valueTextField.setBackground(Color.black);
131           add(valueTextField);
132
133           incrementButton = new JButton();
134           incrementButton.setForeground(Color.black);
135           incrementButton.setBackground(Color.black);
136           incrementButton.setBounds(34,3,25,25);
137           incrementButton.setIcon(loadIcon("images/incbutton.gif"));
138           incrementButton.setDisabledIcon(loadIcon("images/incbuttondisabled.gif"));
139           incrementButton.setPressedIcon(loadIcon("images/incbuttonpressed.gif"));
140           incrementButton.setBorderPainted(false);
141           incrementButton.setFocusPainted(false);
142           incrementButton.addActionListener(this);
143           add(incrementButton);
144
145           decrementButton = new JButton();
146           decrementButton.setForeground(Color.black);
147           decrementButton.setBackground(Color.black);
148           decrementButton.setBounds(34,29,25,25);
149           decrementButton.setIcon(loadIcon("images/decbutton.gif"));
150           decrementButton.setDisabledIcon(loadIcon("images/decbuttondisabled.gif"));
151           decrementButton.setPressedIcon(loadIcon("images/decbuttonpressed.gif"));
152           decrementButton.setBorderPainted(false);
153           decrementButton.setFocusPainted(false);
154           decrementButton.addActionListener(this);
155           add(decrementButton);
156
157           setVisible(true);
158
159           checkButtons();
160        }
161
162
163        /**
164         * Sets the value of the SpinButton object.
165         * @param val The value to set the SpinButton to.  This value must be between
166         * <code>minValue</code> and <code>maxValue</code> (inclusive).
167         */
168        public void setValue(int val)
169        {
170           if (val >= minValue && val <= maxValue)
171           {
172              value = val;
173              intValue = null;
174              intValue = new Integer(value);
175              valueTextField.setText(intValue.toString());
176              checkButtons();
177
178              // Finally, notify our listeners of the change to the value.
179              //fireActionPerformed();
180           }
```

```
181          else
182          {
183             System.out.println("SpinButton::setValue():  Illegal argument: " + val);
184          }
185       }
186
187
188       /**
189        * Gets the value of the SpinButton.
190        * @returns The value of the SpinButton.
191        */
192       public int getValue()
193       {
194          return value;
195       }
196
197
198       /**
199        * Sets the minimum value of the SpinButton object.
200        * @param minVal The minimum value to set the SpinButton to.  This value must be between
201        * <code>0</code> and <code>maxValue</code> (inclusive).
202        */
203       public void setMinValue(int minVal)
204       {
205          if (minVal >= 0 && minVal <= maxValue)
206          {
207             minValue = minVal;
208
209             // If the current value is outside the range, then set it to minValue.
210             if (value < minValue)
211             {
212                value = minValue;
213                intValue = null;
214                intValue = new Integer(value);
215                valueTextField.setText(intValue.toString());
216                checkButtons();
217
218                // Finally, notify our listeners of the change to the value.
219                //fireActionPerformed();
220             }
221          }
222          else
223          {
224             System.out.println("SpinButton::setMinValue():  Illegal argument: " + minVal);
225          }
226       }
227
228
229       /**
230        * Gets the minimum value for the SpinButton.
231        * @returns The minimum value for the SpinButton.
232        */
233       public int getMinValue()
234       {
235          return minValue;
236       }
237
238
239       /**
240        * Sets the maximum value of the SpinButton object.
241        * @param maxVal The maximum value to set the SpinButton to.  This value must be greater
242        * than <code>0</code>.
243        */
244       public void setMaxValue(int maxVal)
245       {
246          if (maxVal >= 0)
247          {
248             maxValue = maxVal;
249
250             // If the current value is outside the range, then set it to maxValue.
251             if (value > maxValue)
252             {
253                value = maxValue;
254                intValue = null;
255                intValue = new Integer(value);
256                valueTextField.setText(intValue.toString());
257                checkButtons();
258
259                // Finally, notify our listeners of the change to the value.
260                //fireActionPerformed();
261             }
262          }
263          else
264          {
265             System.out.println("SpinButton::setMaxValue():  Illegal argument: " + maxVal);
266          }
267       }
268
269
270       /**
```

```
271         * Gets the maximum value for the SpinButton.
272         * @returns The maximum value for the SpinButton.
273         */
274        public int getMaxValue()
275        {
276           return maxValue;
277        }
278
279
280        /**
281         * Enable/disable the buttons according to what the min and max values are.
282         */
283        private void checkButtons()
284        {
285           if (value < maxValue)
286           {
287              incrementButton.setEnabled(true);
288           }
289           else
290           {
291              incrementButton.setEnabled(false);
292           }
293
294
295           if (value > minValue)
296           {
297              decrementButton.setEnabled(true);
298           }
299           else
300           {
301              decrementButton.setEnabled(false);
302           }
303        }
304
305
306        /**
307         * Loads an image corresponding to the given filename from the given directory first.  If
308         * not found, then look for it in the .jar file.  Otherwise, throw an exception.
309         * @param name  The filename for the image.
310         * @returns ImageIcon The image corresponding to the given filename.
311         * @throws NullPointerException
312         */
313        private ImageIcon loadIcon(String name) throws java.lang.NullPointerException
314        {
315           Object icon;
316           String jarName = null;
317           icon = new ImageIcon(name);
318           if (((ImageIcon)icon).getIconWidth() == -1)
319           {
320              jarName = new String("/");
321              jarName = jarName.concat(name);
322
323              try
324              {
325                 icon = new ImageIcon(this.getClass().getResource(jarName));
326              }
327              catch (java.lang.NullPointerException e)
328              {
329                 System.out.println(" ");
330                 System.out.println(" ");
331                 System.out.println("ERROR:  Could not find: " + name);
332                 System.out.println(" ");
333                 System.out.println(" ");
334
335                 throw e;
336              }
337
338              jarName = null;
339           }
340
341           return (ImageIcon)icon;
342        }
343
344
345        /**
346         * See which button was pressed and update the value field accordingly.  If either the min or
347         * max values have been reached, disable the appropriate button.  If the value was already at
348         * the min or max, and we've left that, then enable the appropriate button.
349         */
350        public void actionPerformed(java.awt.event.ActionEvent event)
351        {
352           Object object = event.getSource();
353
354           // Update the value based on which button was pressed.
355           if (object == incrementButton)
356           {
357              value = value + 1;
358              intValue = null;
359              intValue = new Integer(value);
360           }
```

```
361          else if (object == decrementButton)
362          {
363             value = value - 1;
364             intValue = null;
365             intValue = new Integer(value);
366          }
367
368          // Now, display this new value.
369          valueTextField.setText(intValue.toString());
370
371          // Enable/Disable the buttons if needed.
372          checkButtons();
373
374          // Finally, notify our listeners of the change to the value.
375          //fireActionPerformed();
376       }
377    }
```

ATTACHMENT 9

```
1   /**
2    * Filename: LogonDialog.java
3    *
4    * Author: Tom Myers
5    *
6    * Version: 1.0
7    *
8    * Purpose:
9    *
10   * Inputs:
11   *
12   * Outputs:
13   *
14   * Development Environment:  JDK 1.3 (Sun's Java 2 Standard Edition version 1.3) was used.
15   *
16   *
17   * (c) Copyright 2000 Digital Jukebox Technologies LLC.  All Rights Reserved.
18   */
19
20  import java.awt.*;
21  import java.awt.event.*;
22  import java.io.*;
23  import java.security.*;
24  import java.util.Vector;
25  import javax.swing.*;
26  import javax.swing.ImageIcon;
27  import javax.swing.Icon;
28  import javax.swing.Timer;
29  import javax.swing.border.LineBorder;
30
31  import KeyboardPanel.*;
32  import Md5File.*;
33
34  public class LogonDialog extends JDialog implements ActionListener
35  {
36     public static final int OK     = 0;
37     public static final int CANCEL = 1;
38
39     private int iAction = OK;
40     private boolean state = false;
41     private String  strMode = "NONE";
42
43     private Timer       timer = null;
44     private int         iElapsedSec;
45     private int         iTimeoutSec;
46
47     private String      userid = null;
48     private String      passwd = null;
49     private String      newpasswd = null;
50     private boolean     bExpiredPassword = false;
51     private boolean     bChangePassword  = false;
52     private boolean     bChangedPassword = false;
53
54     private JLabel      useridLabel = null;
55     private JLabel      passwdLabel = null;
56
57     private JButton     btn_Logon  = null;
58     private JButton     btn_Cancel = null;
59     private JButton     btn_ChangePassword = null;
60
61     private JTextField     useridTxtField = null;
62     private JPasswordField passwdTxtField = null;
63
64     private KeyboardPanel keyboardPanel = null;
65
66     private String ownerPwd = null;
67     private String adminPwd = null;
68     private String encPasswd = null;
69     private String defaultPassword = null;
70
71
72     public String getUserid()
73     {
74        if (userid != null)
75           return userid;
76        else
77           return "";
78     }
79
80     public String getPassword()
81     {
82        if (passwd != null)
83           return passwd;
84        else
85           return "";
86     }
87
88     public boolean getState()
89     {
90        return state;
```

```
91        }
92
93        public String getMode()
94        {
95           return strMode;
96        }
97
98        public LogonDialog(Frame owner, String title, boolean modal)
99        {
100          super(owner, title, modal);
101          setBounds(-3,-25,1030,800);
102          getContentPane().setLayout(null);
103          setResizable(false);
104
105          timer = new Timer(1000, this);
106          iElapsedSec = 0;
107          iTimeoutSec = 120;
108          timer.start();
109
110          useridLabel = new JLabel("User ID:");
111          useridLabel.setBounds(462,350,80,20);
112          useridLabel.setFont(new Font("SansSerif", Font.BOLD, 12));
113          useridLabel.setForeground(Color.yellow);
114          getContentPane().add(useridLabel);
115
116          useridTxtField = new JTextField();
117          useridTxtField.setBounds(462,370,100,20);
118          useridTxtField.setHorizontalAlignment(JTextField.RIGHT);
119          useridTxtField.setFont(new Font("SansSerif", Font.BOLD, 12));
120          useridTxtField.setForeground(Color.black);
121          useridTxtField.setBackground(Color.white);
122          getContentPane().add(useridTxtField);
123
124          passwdLabel = new JLabel("Password:");
125          passwdLabel.setBounds(462,400,80,20);
126          passwdLabel.setFont(new Font("SansSerif", Font.BOLD, 12));
127          passwdLabel.setForeground(Color.yellow);
128          getContentPane().add(passwdLabel);
129
130          passwdTxtField = new JPasswordField();
131          passwdTxtField.setEchoChar('#');
132          passwdTxtField.setBounds(462,420,100,20);
133          passwdTxtField.setHorizontalAlignment(JTextField.RIGHT);
134          passwdTxtField.setFont(new Font("SansSerif", Font.BOLD, 12));
135          passwdTxtField.setForeground(Color.black);
136          passwdTxtField.setBackground(Color.white);
137          getContentPane().add(passwdTxtField);
138
139          btn_Logon          = new JButton("Logon"); initControl(btn_Logon, 352,500,100,50, false);
140          btn_Cancel         = new JButton("Cancel"); initControl(btn_Cancel, 462,500,100,50, true);
141          btn_ChangePassword = new JButton("ChgPW"); initControl(btn_ChangePassword, 572,500,100,50, false);
142
143          getContentPane().setForeground(Color.yellow);
144          getContentPane().setBackground(Color.black);
145
146          userid    = new String("***");
147          passwd    = new String("###");
148          newpasswd = new String("###");
149
150          keyboardPanel = new KeyboardPanel(0,0,1024,320, useridTxtField, passwdTxtField, userid, passwd, newpasswd,
btn_Logon, btn_ChangePassword);
151          getContentPane().add(keyboardPanel);
152
153          getPasswords();
154
155          setVisible(true);
156       }
157
158       private void getPasswords()
159       {
160          String strFile = "OwnerPwd.ctl";
161          try
162          {
163             BufferedReader in = new BufferedReader(new FileReader(strFile));
164             ownerPwd = in.readLine();
165             in.close();
166
167
168             strFile = "AdminPwd.ctl";
169             in = new BufferedReader(new FileReader(strFile));
170             adminPwd = in.readLine();
171             in.close();
172          }
173          catch (java.io.IOException ioExc)
174          {
175             ioExc.printStackTrace();
176             System.out.println("Error:  Could not read: " + strFile);
177          }
178
179          // The following is to know when we are dealing with a "default" password
```

```
180         // value of "password".  If we encounter this string, then force a change.
181         defaultPassword = encrypt("PASSWORD");
182      }
183
184      private String encrypt(String strText)
185      {
186         MessageDigest md5 = null;
187         String strEncryptedText = null;
188
189         // First, create the message digest.
190         try
191         {
192            md5 = MessageDigest.getInstance("MD5");
193         }
194         catch (java.security.NoSuchAlgorithmException nsaExc)
195         {
196            nsaExc.printStackTrace();
197         }
198
199
200         // Second, encrypt the data.
201         md5.update(strText.getBytes());
202         byte[] hash = md5.digest();
203
204
205         // Finally, return the data as a String object.
206         strEncryptedText = new String(hash);
207
208         return strEncryptedText;
209      }
210
211      private void savePassword()
212      {
213         Md5File md5File = null;
214
215         if (strMode.equalsIgnoreCase("OWNER"))
216         {
217            md5File = new Md5File(passwd, "OwnerPwd.ctl");
218         }
219         else
220         {
221            md5File = new Md5File(passwd, "AdminPwd.ctl");
222         }
223      }
224
225      public void actionPerformed(java.awt.event.ActionEvent e)
226      {
227         Object object = e.getSource();
228
229         if (object == timer)
230         {
231            iElapsedSec = iElapsedSec + 1;
232
233            if (iElapsedSec == iTimeoutSec)
234            {
235               state = false;
236               userid = "";
237               passwd = "";
238
239               setVisible(false);
240            }
241         }
242         else if (object == btn_Cancel)
243         {
244            state = false;
245
246            userid = "";
247            passwd = "";
248
249            setVisible(false);
250         }
251         else if (object == btn_ChangePassword)
252         {
253            bChangedPassword = true;
254            changePassword();
255         }
256         else if (object == btn_Logon)
257         {
258            state = true;
259
260            userid = keyboardPanel.getUserid().toUpperCase();
261            passwd = keyboardPanel.getPassword().toUpperCase();
262
263            if (passwd.equals("###"))
264            {
265               passwd = keyboardPanel.getTextFieldText();
266            }
267
268            encPasswd = encrypt(passwd);
269
```

```
270             if (bExpiredPassword == true)
271             {
272                bExpiredPassword = false;
273                changePassword();
274             }
275             else
276             {
277                if (bChangePassword == true)
278                {
279                   newpasswd = keyboardPanel.getChangePassword();
280
281                   if (passwd.equals(newpasswd))
282                   {
283                      if (userid.equals("OWNER"))
284                         strMode = "OWNER";
285                      else
286                         strMode = "ADMIN";
287
288                      state = true;
289
290                      savePassword();
291
292                      setVisible(false);
293                   }
294                   else
295                   {
296                      strMode = "NONE";
297                      state = false;
298
299                      JOptionPane pane   = new JOptionPane("Passwords did not match...", JOptionPane.
INFORMATION_MESSAGE);
300                      JDialog infoDialog = pane.createDialog(this, "Information");
301                      infoDialog.show();
302
303                      setVisible(false);
304                   }
305                }
306                else
307                {
308                   if (userid.equals("OWNER") && encPasswd.equals(ownerPwd))
309                   {
310                      strMode = "OWNER";
311
312                      if (encPasswd.equals(defaultPassword))
313                      {
314                         bChangedPassword = true;
315                         bExpiredPassword = true;
316
317                         // Give message to user telling them that the password needs to be changed.
318                         JOptionPane pane   = new JOptionPane("Your password needs to be changed. Please enter a 8
character password.", JOptionPane.INFORMATION_MESSAGE);
319                         JDialog infoDialog = pane.createDialog(this, "Password Expired");
320                         infoDialog.show();
321
322                         System.out.println("Password expired...");
323
324                         keyboardPanel.clearPasswordField();
325                      }
326                      else
327                      {
328                         if (bChangedPassword == true)
329                            savePassword();
330
331                         setVisible(false);
332                      }
333                   }
334                   else
335                   {
336                      if (userid.equals("ADMIN") && encPasswd.equals(adminPwd))
337                      {
338                         strMode = "ADMIN";
339
340                         if (encPasswd.equals(defaultPassword))
341                         {
342                            bChangedPassword = true;
343                            bExpiredPassword = true;
344
345                            // Give message to user telling them that the password needs to be changed.
346                            JOptionPane pane   = new JOptionPane("Your password needs to be changed. Please enter a 8
character password.", JOptionPane.INFORMATION_MESSAGE);
347                            JDialog infoDialog = pane.createDialog(this, "Password Expired");
348                            infoDialog.show();
349
350                            System.out.println("Password expired...");
351
352                            keyboardPanel.clearPasswordField();
353                         }
354                         else
355                         {
356                            if (bChangedPassword == true)
```

```
357                                savePassword();
358
359                             setVisible(false);
360                          }
361                       }
362                       else
363                       {
364                          strMode = "NONE";
365                          state = false;
366
367                          JOptionPane pane   = new JOptionPane("Login incorrect.  Please try again...", JOptionPane.
INFORMATION_MESSAGE);
368                          JDialog infoDialog = pane.createDialog(this, "Information");
369                          infoDialog.show();
370
371                          if (bChangedPassword == true)
372                             savePassword();
373
374                          setVisible(false);
375                       }
376                    }
377                 }
378              }
379           }
380        }
381
382        private void changePassword()
383        {
384           JOptionPane pane   = new JOptionPane("Please re-enter your new password.", JOptionPane.INFORMATION_MESSAGE)
;
385           JDialog infoDialog = pane.createDialog(this, "Change Password");
386           infoDialog.show();
387
388           bChangePassword = true;
389
390           keyboardPanel.setChangePasswordTextField();
391        }
392
393        private void initControl(JButton btn, int x1, int y1, int x2, int y2, boolean bEnable)
394        {
395           //btn.setBackground(Color.white);
396           btn.setForeground(Color.blue);
397           btn.setFont(new Font("SansSerif", Font.BOLD, 12));
398           btn.setBounds(x1,y1,x2,y2);
399           btn.setEnabled(bEnable);
400           btn.addActionListener(this);
401           getContentPane().add(btn);
402        }
403     }
```

ATTACHMENT 10

```
/**
 * Filename: ConfirmationDialog.java
 *
 * Author: Tom Myers
 *
 * Version: 1.0
 *
 * Purpose:
 *
 * Inputs:
 *
 * Outputs:
 *
 * Development Environment:  JDK 1.3 (Sun's Java 2 Standard Edition version 1.3) was used.
 *
 *
 * (c) Copyright 2000 Digital Jukebox Technologies LLC.  All Rights Reserved.
 */

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import javax.swing.ImageIcon;
import javax.swing.Icon;
import javax.swing.Timer;
import javax.swing.border.EmptyBorder;
import javax.swing.border.LineBorder;


public class ConfirmationDialog extends JDialog implements ActionListener
{
   public static final int YES_OPTION = 0;
   public static final int NO_OPTION = 1;

   private JLabel label;
   private JTextField txtField;

   private int value = NO_OPTION;
   private JButton yesBtn;
   private JButton noBtn;

   private Timer timer;
   private int iElapsedSec;
   private int iTimeoutSec;


   public ConfirmationDialog(JFrame owner, String title, String text1, String text2)
   {
      super(owner, title, true);


      getContentPane().setLayout(null);
      setBounds(312,284,400,200);
      setResizable(false);
      getContentPane().setForeground(Color.white);
      getContentPane().setBackground(Color.black);


      label = new JLabel(text1, JLabel.CENTER);
      label.setBounds(0,20,400,30);
      label.setFont(new Font("SansSerif", Font.BOLD, 14));
      label.setForeground(Color.yellow);
      label.setBackground(Color.black);
      getContentPane().add(label);


      if (text2 != null)
      {
         txtField = new JTextField(text2);
         txtField.setBounds(25,60,350,20);
         txtField.setEditable(false);
         txtField.setHorizontalAlignment(JTextField.CENTER);
         txtField.setFont(new Font("SansSerif", Font.BOLD, 12));
         txtField.setForeground(Color.white);
         txtField.setBackground(Color.black);
         getContentPane().add(txtField);
      }


      yesBtn = new JButton();
      yesBtn.setIcon(loadIcon("images/yes.gif"));
      yesBtn.setDisabledIcon(loadIcon("images/yesdisabled.gif"));
      yesBtn.setPressedIcon(loadIcon("images/yespressed.gif"));
      yesBtn.setBorderPainted(false);
      yesBtn.setFocusPainted(false);
      yesBtn.setBounds(140,110,50,50);
      yesBtn.addActionListener(this);
      getContentPane().add(yesBtn);

```

```
92          noBtn = new JButton();
93          noBtn.setIcon(loadIcon("images/no.gif"));
94          noBtn.setDisabledIcon(loadIcon("images/nodisabled.gif"));
95          noBtn.setPressedIcon(loadIcon("images/nopressed.gif"));
96          noBtn.setBorderPainted(false);
97          noBtn.setFocusPainted(false);
98          noBtn.setBounds(210,110,50,50);
99          noBtn.addActionListener(this);
100         getContentPane().add(noBtn);
101
102         timer = new Timer(1000, this);
103         iElapsedSec = 0;
104         iTimeoutSec = 120;
105         timer.start();
106
107
108         setVisible(true);
109      }
110
111
112      public void actionPerformed(java.awt.event.ActionEvent event)
113      {
114         Object object = event.getSource();
115
116         if (object == timer)
117         {
118            iElapsedSec = iElapsedSec + 1;
119
120            if (iElapsedSec == iTimeoutSec)
121            {
122               value = NO_OPTION;
123               setVisible(false);
124            }
125         }
126         else if (object == noBtn)
127         {
128            value = NO_OPTION;
129            setVisible(false);
130         }
131         else if (object == yesBtn)
132         {
133            value = YES_OPTION;
134            setVisible(false);
135         }
136      }
137
138
139      public int getValue()
140      {
141         return value;
142      }
143
144
145      private ImageIcon loadIcon(String name) throws java.lang.NullPointerException
146      {
147         Object icon;
148         String jarName = null;
149         icon = new ImageIcon(name);
150         if (((ImageIcon)icon).getIconWidth() == -1)
151         {
152            jarName = new String("/");
153            jarName = jarName.concat(name);
154
155            try
156            {
157               icon = new ImageIcon(this.getClass().getResource(jarName));
158            }
159            catch (java.lang.NullPointerException e)
160            {
161               System.out.println(" ");
162               System.out.println(" ");
163               System.out.println("ERROR:  Could not find: " + name);
164               System.out.println(" ");
165               System.out.println(" ");
166
167               throw e;
168            }
169
170            jarName = null;
171         }
172
173         return (ImageIcon)icon;
174      }
175   }
```

ATTACHMENT 11

```
/**
 * Filename: Md5File.java
 *
 * Author: Tom Myers
 *
 * Version: 1.0
 *
 * Purpose: Used to create the encrypted password files for the "admin" and "owner" userids
 *          that are used to restrict access to the administrator panel.
 *
 * Inputs:
 *
 * Outputs:
 *
 * Development Environment:  JDK 1.3 (Sun's Java 2 Standard Edition version 1.3) was used.
 *
 *
 * (c) Copyright 2000 Digital Jukebox Technologies LLC.  All Rights Reserved.
 */

import java.io.*;
import java.security.*;

public class Md5File
{
   private MessageDigest md5 = null;
   private File md5File = null;
   private BufferedWriter out = null;

   public Md5File(String strToEncrypt, String strFilename)
   {
      // First, create the message digest.
      try
      {
         md5 = MessageDigest.getInstance("MD5");
      }
      catch (java.security.NoSuchAlgorithmException nsaExc)
      {
         nsaExc.printStackTrace();
      }


      // Second, encrypt the data.
      md5.update(strToEncrypt.getBytes());
      byte[] hash = md5.digest();


      // Finally, create the file and save the hash contents to it.
      try
      {
         md5File = new File(strFilename);
         out = new BufferedWriter(new FileWriter(md5File));

         String s1 = new String(hash);
         out.write(s1, 0, s1.length());
         out.newLine();

         out.flush();
         out.close();
      }
      catch (java.io.IOException ioExc)
      {
         ioExc.printStackTrace();
      }
   }

   static public void main(String args[])
   {
      if (args.length == 2)
      {
         Md5File md5File = new Md5File(args[0], args[1]);
      }
      else
      {
         System.out.println(" ");
         System.out.println("Please enter a string to encrypt and a file to save it to.");
         System.out.println("  e.g. java md5File TextToEncrypt C:\\password.ctl");
         System.out.println(" ");
      }
   }
}
```

ATTACHMENT 12

```
/**
 * Filename: JMFMgr.java
 *
 * Author: Tom Myers
 *
 * Version: 1.0
 *
 * Purpose: This is the "all-java" equivalent of the WinAmpMgr class and can be used to
 *          perform the same functions as WinAmpMgr if Winamp is not to be used to render
 *          (or play) the MP3/Wav files for the application
 *
 * Inputs:
 *
 * Outputs:
 *
 * Development Environment:  JDK 1.3 (Sun's Java 2 Standard Edition version 1.3) was used.
 *         The javax.media package (Java Media Framework 2.0) is required for this class.
 *
 *
 * (c) Copyright 2000 Digital Jukebox Technologies LLC.  All Rights Reserved.
 */

import java.io.*;
import java.net.URL;
import java.net.MalformedURLException;
import javax.swing.*;
import javax.media.*;
import javax.media.bean.playerbean.*;
import javax.media.Renderer;
import javax.media.control.TrackControl;
import javax.media.control.BufferControl;
import javax.media.format.*;
import javax.media.protocol.*;

import PlayerMgr.*;
import TreeMgr.*;

public class JmfMgr extends PlayerMgr implements ControllerListener
{
   protected File        file;
   protected URL         url;
   protected Time        pauseTime;
   protected Time        time;
   protected GainControl gainControl;
   protected DataSource  dataSource;
   protected Player      player;

   public JmfMgr()
   {
      super();

      file        = null;
      url         = null;
      pauseTime   = null;
      time        = null;
      gainControl = null;
      dataSource  = null;
      player      = null;
   }


   public int getOutputTime()
   {
      if (player != null)
      {
         time = player.getMediaTime();
         return (int)time.getSeconds();
      }
      else
         return 0;
   }


   public int getSongLength()
   {
      Time songTime = player.getDuration();
      int  iTime = 0;

      if (songTime != Duration.DURATION_UNKNOWN)
      {
         iTime = (int)songTime.getSeconds();
      }
      return iTime;
   }


   public void pressPause()
   {
      if (iStatus == PAUSED)
      {
```

```
91              iStatus = PLAYING;
92              player.setMediaTime(pauseTime);
93              player.start();
94           }
95           else
96           {
97              iStatus = PAUSED;
98              pauseTime = player.getMediaTime();
99              player.stop();
100          }
101          return;
102       }
103
104
105       public void pressStop()
106       {
107          iStatus = STOPPED;
108
109          if (player != null)
110          {
111             player.stop();
112             player.deallocate();
113          }
114
115          return;
116       }
117
118       public void cleanUp()
119       {
120          if (player != null)
121          {
122             player.close();
123             player = null;
124          }
125       }
126
127       public void play(PlayListEntry mp3)
128       {
129          iStatus = PLAYING;
130
131          // Increase our prioity so the player will have a high priority.
132          this.setPriority(9);
133
134          String strMp3 = mp3.getMp3Path();
135
136          try
137          {
138             file = new File(strMp3);
139             if (file.exists())
140             {
141                // Remove references to the previously played song.
142                if (player != null)
143                   player.close();
144
145                url = new URL("file:///" + strMp3);
146
147                dataSource = Manager.createDataSource(url);
148
149                player = Manager.createRealizedPlayer(dataSource);
150
151                     Control cs[] = player.getControls();
152                     Object owner;
153
154                     for (int i = 0; i < cs.length; i++)
155             {
156                if (cs[i] instanceof Owned && cs[i] instanceof BufferControl)
157                {
158                            owner = ((Owned)cs[i]).getOwner();
159
160                   if (owner != null)
161                   {
162                      if (owner instanceof Renderer)
163                      {
164                         ((BufferControl)cs[i]).setBufferLength(4000);
165                      }
166                            }
167                  }
168               }
169
170                player.addControllerListener(this);
171                player.start();
172
173                gainControl = player.getGainControl();
174                setVolume(iVolume);
175
176                iSongLength = getSongLength();
177
178                mp3.decrementQueuedCnt();
179                mp3.incrementPlayedCnt();
180
```

```
181                if (mp3.getPaidQueuedCnt() > 0)
182                   mp3.decrementPaidQueuedCnt();
183
184                strCurrentSong = mp3.getMp3Path();
185                currentPlayListObj = mp3;
186             }
187          }
188          catch (java.net.MalformedURLException e)
189          {
190             System.out.println("Could not form URL for: " + strMp3);
191          }
192          catch (javax.media.NoPlayerException noPlayerExcptn)
193          {
194             System.out.println("NoPlayerException occurred for: " + strMp3);
195          }
196          catch (java.io.IOException ioExcptn)
197          {
198             System.out.println("IOException occurred for: " + strMp3);
199          }
200          catch (javax.media.CannotRealizeException realizeExcptn)
201          {
202             System.out.println("Cannot realize player for: " + strMp3);
203          }
204          catch (Exception e0)
205          {
206             System.err.println("Exception: " + e0);
207             System.out.println(" ");
208             e0.printStackTrace();
209          }
210       }
211
212       public void controllerUpdate(javax.media.ControllerEvent event)
213       {
214          if (event instanceof EndOfMediaEvent)
215          {
216             iStatus = STOPPED;
217             player.deallocate();
218          }
219       }
220
221       public void setVolume(int iVol)
222       {
223          if (iVol < 0)
224             iVol = 0;
225
226          if (iVol > 100)
227             iVol = 100;
228
229          iVolume = iVol;
230
231          if (gainControl != null)
232          {
233             float fVol = ((float)iVol) / 100;
234             gainControl.setLevel(fVol);
235          }
236       }
237
238
239       public void run()
240       {
241          // Run forever.
242          while (true)
243          {
244             // While a song is playing or paused, keep track of its progress.
245             while (iStatus != STOPPED)
246             {
247                iTimeRemaining = iSongLength - getOutputTime();
248
249                try
250                {
251                   sleep(1000);
252                }
253                catch (Exception excptn)
254                {
255                   System.out.println(excptn.toString());
256                }
257             }
258
259             // If song finished, or was stopped by the user, see if there are more to play.
260             if ( (!playListVector.isEmpty()) && (!bLockOnQueue) && (!bFirstTime) )
261             {
262                play(((PlayListEntry)playListVector.firstElement()));
263
264                removeFromPlayList(((PlayListEntry)playListVector.firstElement()));
265             }
266             else
267             {
268                // Since there's nothing to do, lower our priority to the default.
269                this.setPriority(5);
270
```

```
271                 // Go to sleep, waking up one a sec. to see if there are any songs to play.
272                 try
273                 {
274                    sleep(1000);
275                 }
276                 catch (Exception excptn)
277                 {
278                    System.out.println(excptn.toString());
279                 }
280              }
281           }
282        }
283     }
```

ATTACHMENT 13

```
/**
 * Filename: MyListRenderer.java
 *
 * Author: Tom Myers
 *
 * Version: 1.0
 *
 * Purpose: Used to custom render JList items based upon PlayListEntry objects as list items
 *
 * Inputs:
 *
 * Outputs:
 *
 * Development Environment:  JDK 1.3 (Sun's Java 2 Standard Edition version 1.3) was used.
 *
 *
 * (c) Copyright 2000 Digital Jukebox Technologies LLC.  All Rights Reserved.
 */

import java.awt.*;
import java.awt.Component;
import java.net.URL;

import javax.swing.*;

import TreeMgr.*;

public class MyListRenderer extends DefaultListCellRenderer
{
    /** Icon used to show visible status. */
    transient protected Icon level_0_Icon;
    transient protected Icon level_1_Icon;
    transient protected Icon level_2_Icon;
    transient protected Icon level_3_Icon;
    transient protected Icon queued_Icon;

    transient protected Icon sel_level_0_Icon;
    transient protected Icon sel_level_1_Icon;
    transient protected Icon sel_level_2_Icon;
    transient protected Icon sel_level_3_Icon;
    transient protected Icon sel_queued_Icon;

    private int iLevel_0_Floor;
    private int iLevel_1_Floor;
    private int iLevel_2_Floor;

    private boolean bShowQueued = true;


    /**
      * Returns a new instance of MyListCellRenderer.  Alignment is
      * set to left aligned.
      */
    public MyListRenderer()
    {
       this(0, 5, 10, true);
    }


    /**
      * Returns a new instance of MyListCellRenderer.  Alignment is
      * set to left aligned.
      */
    public MyListRenderer(int iLev0, int iLev1, int iLev2, boolean bShow)
    {
       bShowQueued = bShow;

       iLevel_0_Floor = iLev0;
       iLevel_1_Floor = iLev1;
       iLevel_2_Floor = iLev2;

       level_0_Icon  = loadIcon("images/level_0.gif");
       level_1_Icon  = loadIcon("images/level_1.gif");
       level_2_Icon  = loadIcon("images/level_2.gif");
       level_3_Icon  = loadIcon("images/level_3.gif");
       queued_Icon   = loadIcon("images/queued.gif");

       sel_level_0_Icon  = loadIcon("images/level_0_selected.gif");
       sel_level_1_Icon  = loadIcon("images/level_1_selected.gif");
       sel_level_2_Icon  = loadIcon("images/level_2_selected.gif");
       sel_level_3_Icon  = loadIcon("images/level_3_selected.gif");
       sel_queued_Icon   = loadIcon("images/queued_selected.gif");
    }


    /**
      * Configures the renderer based on the passed in components.
      * The value is set from messaging value with toString().
      * The foreground color is set based on the selection and the icon
      * is set based on on leaf and expanded.
```

```
91          */
92         public java.awt.Component getListCellRendererComponent(JList   list,
93                                                                 Object  value,
94                                                                 int     index,
95                                                                 boolean isSelected,
96                                                                 boolean cellHasFocus)
97         {
98            super.getListCellRendererComponent(list,
99                                               value,
100                                              index,
101                                              isSelected,
102                                              cellHasFocus);
103
104           setText(getTitle(value));
105
106           if (isSelected)
107           {
108              if (bShowQueued == true && isQueued(value))
109              {
110                 setIcon(sel_queued_Icon);
111              }
112              else
113              {
114                 int iLevel = getPlayedLevel(value);
115
116                 if (iLevel == 0)
117                    setIcon(sel_level_0_Icon);
118                 else if (iLevel == 1)
119                    setIcon(sel_level_1_Icon);
120                 else if (iLevel == 2)
121                    setIcon(sel_level_2_Icon);
122                 else
123                    setIcon(sel_level_3_Icon);
124              }
125           }
126           else
127           {
128              if (bShowQueued == true && isQueued(value))
129              {
130                 setIcon(queued_Icon);
131              }
132              else
133              {
134                 int iLevel = getPlayedLevel(value);
135
136                 if (iLevel == 0)
137                    setIcon(level_0_Icon);
138                 else if (iLevel == 1)
139                    setIcon(level_1_Icon);
140                 else if (iLevel == 2)
141                    setIcon(level_2_Icon);
142                 else
143                    setIcon(level_3_Icon);
144              }
145           }
146
147           return this;
148        }
149
150        protected String getTitle(Object value)
151        {
152           byte nameIdx = 0;
153
154           if (value instanceof PlayListEntry)
155           {
156              nameIdx = ((PlayListEntry)value).getNameIdx();
157           }
158           String title = value.toString();
159
160           return title.substring(nameIdx, title.length() - 4);
161        }
162
163
164        protected boolean isQueued(Object value)
165        {
166           boolean bQueued = false;
167
168           if (value instanceof PlayListEntry)
169           {
170              int iQueuedCnt = ((PlayListEntry)value).getQueuedCnt();
171
172              if (iQueuedCnt > 0)
173                 bQueued = true;
174           }
175
176           return bQueued;
177        }
178
179
180        protected int getPlayedLevel(Object value)
```

```
181         {
182            int paidCnt = 0;
183            int iLevel  = 0;
184
185            if (value instanceof PlayListEntry)
186            {
187               paidCnt = ((PlayListEntry)value).getPaidCnt();
188
189               if (paidCnt == iLevel_0_Floor)
190                  iLevel = 0;
191               else if (paidCnt > iLevel_0_Floor && paidCnt < iLevel_1_Floor)
192                  iLevel = 1;
193               else if (paidCnt >= iLevel_1_Floor && paidCnt < iLevel_2_Floor)
194                  iLevel = 2;
195               else
196                  iLevel = 3;
197            }
198
199            return iLevel;
200         }
201
202
203         private ImageIcon loadIcon(String name) throws java.lang.NullPointerException
204         {
205            Object icon;
206            String jarName = null;
207            icon = new ImageIcon(name);
208            if (((ImageIcon)icon).getIconWidth() == -1)
209            {
210               jarName = new String("/");
211               jarName = jarName.concat(name);
212
213               try
214               {
215                  icon = new ImageIcon(this.getClass().getResource(jarName));
216               }
217               catch (java.lang.NullPointerException e)
218               {
219                  System.out.println(" ");
220                  System.out.println(" ");
221                  System.out.println("ERROR:  Could not find: " + name);
222                  System.out.println(" ");
223                  System.out.println(" ");
224
225                  throw e;
226               }
227
228               jarName = null;
229            }
230
231            return (ImageIcon)icon;
232         }
233      }
```

ATTACHMENT 14

```
/**
 * Filename: MyRenderer.java
 *
 * Author: Tom Myers
 *
 * Version: 1.0
 *
 * Purpose: Used to custom render the tree cell components
 *
 * Inputs:
 *
 * Outputs:
 *
 * Development Environment:  JDK 1.3 (Sun's Java 2 Standard Edition version 1.3) was used.
 *
 *
 * (c) Copyright 2000 Digital Jukebox Technologies LLC.  All Rights Reserved.
 */

import java.awt.*;
import java.awt.Component;
import java.net.URL;

import javax.swing.JTree;
import javax.swing.tree.DefaultTreeCellRenderer;
import javax.swing.tree.DefaultMutableTreeNode;
import javax.swing.ImageIcon;
import javax.swing.Icon;

import TreeMgr.*;

public class MyRenderer extends DefaultTreeCellRenderer
{
    /** Icon used to show non-played leaf nodes. */
    transient protected Icon queuedLeafIcon;

    /** Icon used to show parent nodes. */
    transient protected Icon folderIcon;

    /** Icon used to show the currently playing node. */
    transient protected Icon playingIcon;

    /** Icon used to blank out Java look and feel node icon. */
    transient protected Icon blankIcon;


    /** Icon used to show visible status. */
    transient protected Icon level_0_Icon;
    transient protected Icon level_1_Icon;
    transient protected Icon level_2_Icon;
    transient protected Icon level_3_Icon;
    transient protected Icon queued_Icon;

    private int iLevel_0_Floor;
    private int iLevel_1_Floor;
    private int iLevel_2_Floor;


    /**
      * Returns a new instance of MyRenderer.  Alignment is
      * set to left aligned.
      */
    public MyRenderer()
    {
       this(0, 5, 10);
    }


    /**
      * Returns a new instance of MyRenderer.  Alignment is
      * set to left aligned.
      */;
    public MyRenderer(int iLev0, int iLev1, int iLev2)
    {
       iLevel_0_Floor = iLev0;
       iLevel_1_Floor = iLev1;
       iLevel_2_Floor = iLev2;

       level_0_Icon = loadIcon("images/level_0.gif");
       level_1_Icon = loadIcon("images/level_1.gif");
       level_2_Icon = loadIcon("images/level_2.gif");
       level_3_Icon = loadIcon("images/level_3.gif");

       queuedLeafIcon = loadIcon("images/queued.gif");
       folderIcon     = loadIcon("images/folder.gif");
       playingIcon    = loadIcon("images/playing.gif");
       blankIcon      = loadIcon("images/blank.gif");

       setOpenIcon(blankIcon);
       setBackgroundNonSelectionColor(Color.black);
```

```
91          setBackgroundSelectionColor(new Color(0,0,128));
92          setTextNonSelectionColor(Color.white);
93          setTextSelectionColor(Color.white);
94       }
95
96
97
98       /**
99         * Configures the renderer based on the passed in components.
100        * The value is set from messaging value with toString().
101        * The foreground color is set based on the selection and the icon
102        * is set based on on leaf and expanded.
103        */
104      public java.awt.Component getTreeCellRendererComponent(JTree   tree,
105                                                             Object  value,
106                                                             boolean sel,
107                                                             boolean expanded,
108                                                             boolean leaf,
109                                                             int     row,
110                                                             boolean hasFocus)
111      {
112         super.getTreeCellRendererComponent(tree,
113                                            value,
114                                            sel,
115                                            expanded,
116                                            leaf,
117                                            row,
118                                            hasFocus);
119
120         // If the mp3 corresponding to this leaf node has already been played, then
121         // use the appropriate icon.
122         if (isPlayListEntry(value))
123         {
124            if (isQueued(value))
125            {
126              setIcon(queuedLeafIcon);
127            }
128            else
129            {
130               if (isCurrentlyPlayingSong(value))
131               {
132                  setIcon(playingIcon);
133               }
134               else
135               {
136                  int iLevel = getPlayedLevel(value);
137
138                  if (iLevel == 0)
139                     setIcon(level_0_Icon);
140                  else if (iLevel == 1)
141                     setIcon(level_1_Icon);
142                  else if (iLevel == 2)
143                     setIcon(level_2_Icon);
144                  else
145                     setIcon(level_3_Icon);
146               }
147            }
148         }
149         else
150         {
151            setIcon(folderIcon);
152         }
153
154         return this;
155      }
156
157
158
159      protected boolean isQueued(Object value)
160      {
161         DefaultMutableTreeNode node = (DefaultMutableTreeNode)value;
162
163         if (node.getUserObject() instanceof PlayListEntry)
164         {
165            PlayListEntry nodeObject = (PlayListEntry)(node.getUserObject());
166
167            if (nodeObject.getQueuedCnt() > 0 || nodeObject.getPaidQueuedCnt() > 0)
168               return true;
169         }
170
171         return false;
172      }
173
174
175      protected boolean isPlayListEntry(Object value)
176      {
177         DefaultMutableTreeNode node = (DefaultMutableTreeNode)value;
178
179         if (node.getUserObject() instanceof PlayListEntry)
180            return true;
```

```
181          else
182             return false;
183       }
184
185
186       protected int getPlayedLevel(Object value)
187       {
188          int paidCnt = 0;
189          int iLevel  = 0;
190
191          DefaultMutableTreeNode node = (DefaultMutableTreeNode)value;
192
193          if (node.getUserObject() instanceof PlayListEntry)
194          {
195             PlayListEntry nodeObject = (PlayListEntry)(node.getUserObject());
196
197             paidCnt = nodeObject.getPaidCnt();
198
199             if (paidCnt == iLevel_0_Floor)
200                iLevel = 0;
201             else if (paidCnt > iLevel_0_Floor && paidCnt < iLevel_1_Floor)
202                iLevel = 1;
203             else if (paidCnt >= iLevel_1_Floor && paidCnt < iLevel_2_Floor)
204                iLevel = 2;
205             else
206                iLevel = 3;
207          }
208
209          return iLevel;
210       }
211
212
213       protected boolean isCurrentlyPlayingSong(Object value)
214       {
215          try
216          {
217             DefaultMutableTreeNode node = (DefaultMutableTreeNode)value;
218
219             if (node.getUserObject() instanceof PlayListEntry)
220             {
221                PlayListEntry nodeObject = (PlayListEntry)(node.getUserObject());
222
223                String strCurrentSong = nodeObject.getCurrPlayingSong();
224
225                if (strCurrentSong != null)
226                   if (nodeObject.getMp3Path().equalsIgnoreCase(strCurrentSong))
227                      return true;
228             }
229          }
230          catch (java.lang.NullPointerException e)
231          {
232             return false;
233          }
234
235          return false;
236       }
237
238
239
240
241       private ImageIcon loadIcon(String name) throws java.lang.NullPointerException
242       {
243          Object icon;
244          String jarName = null;
245          icon = new ImageIcon(name);
246          if (((ImageIcon)icon).getIconWidth() == -1)
247          {
248             jarName = new String("/");
249             jarName = jarName.concat(name);
250
251             try
252             {
253                icon = new ImageIcon(this.getClass().getResource(jarName));
254             }
255             catch (java.lang.NullPointerException e)
256             {
257                System.out.println(" ");
258                System.out.println(" ");
259                System.out.println("ERROR:  Could not find: " + name);
260                System.out.println(" ");
261                System.out.println(" ");
262
263                throw e;
264             }
265
266             jarName = null;
267          }
268
269          return (ImageIcon)icon;
270       }
```

```
272
273     }
```

ATTACHMENT 15

```
/**
 * Filename: CustomFileView.java
 *
 * Author: Tom Myers
 *
 * Version: 1.0
 *
 * Purpose:
 *
 * Inputs:
 *
 * Outputs:
 *
 * Development Environment:  JDK 1.3 (Sun's Java 2 Standard Edition version 1.3) was used.
 *
 *
 * (c) Copyright 2000 Digital Jukebox Technologies LLC.  All Rights Reserved.
 */

import javax.swing.filechooser.FileView;
import javax.swing.ImageIcon;
import javax.swing.Icon;
import java.io.*;


class CustomFileView extends FileView
{
   private Icon playListIcon  = new ImageIcon("playlist.gif");
   private Icon directoryIcon = new ImageIcon("fldr.gif");
   private Icon fileIcon      = new ImageIcon("file.gif");

   public String getName(File f) { return null; }
   public String getDescription(File f) { return null; }
   public String getTypeDescription(File f) { return null; }

   public Icon getIcon(File f)
   {
      Icon icon = null;

      if (isPlayList(f))
         icon = playListIcon;
      else
         if (f.isDirectory())
            icon = directoryIcon;
         else
            icon = fileIcon;
      return icon;
   }

   public Boolean isTraversable(File f)
   {
      return new Boolean(true);
   }

   private boolean isPlayList(File f)
   {
      String suffix = getSuffix(f);
      boolean isPlayList = false;

      if (suffix != null)
      {
         isPlayList = suffix.equals("pl");
      }

      return isPlayList;
   }

   private String getSuffix(File file)
   {
      String filestr = file.getPath();
      String suffix = null;
      int i = filestr.lastIndexOf('.');

      if (i > 0 && i < filestr.length())
      {
         suffix = filestr.substring(i+1).toLowerCase();
      }
      return suffix;
   }
}
```

ATTACHMENT 16

```
/**
 * Filename: WinampFilter.java
 *
 * Author: Tom Myers
 *
 * Version: 1.0
 *
 * Purpose:
 *
 * Inputs:
 *
 * Outputs:
 *
 * Development Environment:  JDK 1.3 (Sun's Java 2 Standard Edition version 1.3) was used.
 *
 *
 * (c) Copyright 2000 Digital Jukebox Technologies LLC.  All Rights Reserved.
 */

import javax.swing.filechooser.FileFilter;
import java.io.File;

class WinampFilter extends FileFilter
{
   public boolean accept(File f)
   {
      boolean accept = f.isDirectory();

      if (!accept)
      {
         if (f.toString().toLowerCase().indexOf("winamp.exe") != -1)
            accept = true;
      }
      return accept;
   }

   public String getDescription()
   {
      return "Winamp Program File (Winamp.exe)";
   }
}
```

ATTACHMENT 17

```
/**
 * Filename: AddPathDialog.java
 *
 * Author: Tom Myers
 *
 * Version: 1.0
 *
 * Purpose:
 *
 * Inputs:
 *
 * Outputs:
 *
 * Development Environment:  JDK 1.3 (Sun's Java 2 Standard Edition version 1.3) was used.
 *
 *
 * (c) Copyright 2000 Digital Jukebox Technologies LLC.  All Rights Reserved.
 */

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.Vector;
import java.util.Enumeration;

import javax.swing.*;
import javax.swing.ImageIcon;
import javax.swing.Icon;

public class AddPathDialog extends JDialog implements ActionListener
{
   private JLabel      addLabel1  = null;
   private JComboBox   addCombo   = null;
   private Vector      addVector  = null;

   private JLabel      addLabel2    = null;
   private JTextField  addBrowseTxt = null;
   private JButton     addBrowseBtn = null;

   private JLabel      addLabel3    = null;
   private JTextArea   addTypesText = null;

   private JLabel      addLabel4 = null;
   private JLabel      addLabel5 = null;

   private JButton     addSearchBtn = null;
   private JButton     addDoneBtn   = null;
   private JButton     addCancelBtn = null;

   private JLabel      addLabel6  = null;

   private boolean     state  = false;
   private Vector      vector = null;

   private int         comboIndex = 0; // For keeping track of selected item in combo box.


   public AddPathDialog(Frame owner, String title, boolean modal, int x, int y)
   {
      super(owner, title, modal);
      setBounds(x, y, 320, 240);
      getContentPane().setLayout(null);

      addCombo = new JComboBox();
      addBrowseTxt = new JTextField("All Folders");
      addBrowseBtn = new JButton("Browse...");

      initialize();

      addCombo.setMaximumRowCount(10);
      addCombo.setBounds(115,10,85,25);
      //addCombo.setForeground(Color.white);
      //addCombo.setBackground(Color.black);
      addCombo.setSelectedIndex(comboIndex);
      getContentPane().add(addCombo);
      addCombo.addItemListener(
         new ItemListener()
         {
            public void itemStateChanged(ItemEvent e)
            {
               comboIndex = addCombo.getSelectedIndex();

               itemStateChanged_doWork();
            }
         });

      addCombo.addKeyListener(
         new KeyListener()
         {
            public void keyPressed(java.awt.event.KeyEvent event)
```

```
91              {
92                 char ch = event.getKeyChar();
93
94                 if (ch == java.awt.event.KeyEvent.VK_ESCAPE)
95                 {
96                    addCancelBtn_doWork();
97                 }
98
99                 if (ch == java.awt.event.KeyEvent.VK_ENTER)
100                {
101                   addSearchBtn_doWork();
102                }
103             }
104
105             public void keyReleased(java.awt.event.KeyEvent event) { }
106             public void keyTyped(java.awt.event.KeyEvent event) { }
107          });
108
109
110       addLabel1 = new JLabel("Look for Songs In:");
111       addLabel1.setBounds(5,15,110,15);
112       getContentPane().add(addLabel1);
113
114       addLabel2 = new JLabel("Searching In:");
115       addLabel2.setBounds(5,50,110,15);
116       getContentPane().add(addLabel2);
117
118       addBrowseTxt.setForeground(Color.white);
119       addBrowseTxt.setBackground(Color.black);
120       addBrowseTxt.setEditable(false);
121       addBrowseTxt.setBounds(90,45,110,25);
122       getContentPane().add(addBrowseTxt);
123
124       addBrowseBtn.setBounds(205,45,90,25);
125       addBrowseBtn.setForeground(Color.blue);
126       addBrowseBtn.setEnabled(false);
127       getContentPane().add(addBrowseBtn);
128       addBrowseBtn.addActionListener(this);
129
130       addLabel3 = new JLabel("File types:");
131       addLabel3.setBounds(5,85,80,15);
132       getContentPane().add(addLabel3);
133
134       addTypesText = new JTextArea(".MP3            .WAV");
135       addTypesText.setBounds(90,80,110,35);
136       addTypesText.setForeground(Color.white);
137       addTypesText.setBackground(Color.black);
138       addTypesText.setEditable(false);
139       addTypesText.setLineWrap(true);
140       getContentPane().add(addTypesText);
141
142       addLabel4 = new JLabel("Note: Only the locations of these files will be recorded");
143       addLabel4.setBounds(5,130,315,15);
144       getContentPane().add(addLabel4);
145
146       addLabel5 = new JLabel("by MP3Jukebox. No changes will be made to these files.");
147       addLabel5.setBounds(5,145,315,15);
148       getContentPane().add(addLabel5);
149
150       addSearchBtn = new JButton("Search");
151       addSearchBtn.setActionCommand("Search");
152       addSearchBtn.setBounds(25,170,80,25);
153       addSearchBtn.setForeground(Color.blue);
154       getContentPane().add(addSearchBtn);
155       addSearchBtn.addActionListener(this);
156
157       addDoneBtn = new JButton("Done");
158       addDoneBtn.setActionCommand("Done");
159       addDoneBtn.setBounds(110,170,80,25);
160       addDoneBtn.setForeground(Color.blue);
161       getContentPane().add(addDoneBtn);
162       addDoneBtn.addActionListener(this);
163
164       addCancelBtn = new JButton("Cancel");
165       addCancelBtn.setActionCommand("Cancel");
166       addCancelBtn.setBounds(195,170,80,25);
167       addCancelBtn.setForeground(Color.blue);
168       getContentPane().add(addCancelBtn);
169       addCancelBtn.addActionListener(this);
170
171       setVisible(true);
172    }
173
174    public void actionPerformed(java.awt.event.ActionEvent event)
175    {
176       Object object = event.getSource();
177
178       if (object == addCancelBtn || object == addDoneBtn)
179       {
180          addCancelBtn_doWork();
```

```
181         }
182         else if (object == addSearchBtn)
183         {
184            addSearchBtn_doWork();
185         }
186         else if (object == addBrowseBtn)
187         {
188            addBrowseBtn_doWork();
189         }
190      }
191
192      public void initialize()
193      {
194         addVector = new Vector();
195         addVector.addElement("All Drives");
196
197         String driveList[] = new String[24];
198         driveList[0]  = new String("C:\\");
199         driveList[1]  = new String("D:\\");
200         driveList[2]  = new String("E:\\");
201         driveList[3]  = new String("F:\\");
202         driveList[4]  = new String("G:\\");
203         driveList[5]  = new String("H:\\");
204         driveList[6]  = new String("I:\\");
205         driveList[7]  = new String("J:\\");
206         driveList[8]  = new String("K:\\");
207         driveList[9]  = new String("L:\\");
208         driveList[10] = new String("M:\\");
209         driveList[11] = new String("N:\\");
210         driveList[12] = new String("O:\\");
211         driveList[13] = new String("P:\\");
212         driveList[14] = new String("Q:\\");
213         driveList[15] = new String("R:\\");
214         driveList[16] = new String("S:\\");
215         driveList[17] = new String("T:\\");
216         driveList[18] = new String("U:\\");
217         driveList[19] = new String("V:\\");
218         driveList[20] = new String("W:\\");
219         driveList[21] = new String("X:\\");
220         driveList[22] = new String("Y:\\");
221         driveList[23] = new String("Z:\\");
222
223         File tmpFile = null;
224
225         // Recurse the drives that were found.
226         int j = 0;
227         try
228         {
229            for (j = 0; j < driveList.length; j++)
230            {
231               tmpFile = new File(driveList[j]);
232
233               if (tmpFile.exists())
234               {
235                  addVector.addElement(tmpFile);
236               }
237            }
238         }
239         catch(NullPointerException excptn)
240         {
241            System.out.println("Drive doesn't exist: " + driveList[j]);
242         }
243
244         addCombo.setModel(new DefaultComboBoxModel(addVector));
245
246         if (comboIndex <= j)
247         {
248            itemStateChanged_doWork();
249         }
250         else
251         {
252            comboIndex = 0;
253            addCombo.setSelectedIndex(comboIndex);
254            addBrowseTxt.setText("All Folders");
255            addBrowseBtn.setEnabled(false);
256         }
257
258         state  = false;
259         vector = null;
260      }
261
262      public boolean getState()
263      {
264         return state;
265      }
266
267      public Vector getVector()
268      {
269         return vector;
270      }
```

```
272     private void addCancelBtn_doWork()
273     {
274        state = false;
275        setVisible(false);
276     }
277
278     private void addSearchBtn_doWork()
279     {
280        // Show the "Scanning, please wait..." message to the user...
281        addLabel6 = new JLabel("Scanning for songs, please wait...", JLabel.CENTER);
282        addLabel6.setBounds(60,196,200,15);
283        getContentPane().add(addLabel6);
284
285        if (addBrowseTxt.getText().equalsIgnoreCase("All Folders"))
286        {
287           vector = addVector;
288           vector.removeElementAt(0);
289        }
290        else
291        {
292           File file = new File(addBrowseTxt.getText());
293           vector = new Vector();
294           vector.addElement(file);
295        }
296        setVisible(false);
297        state = true;
298     }
299
300     private void addBrowseBtn_doWork()
301     {
302        JFileChooser  chooser;
303        chooser = new JFileChooser(addBrowseTxt.getText());
304        chooser.setApproveButtonText("Add");
305        chooser.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);
306
307        int chooserState = chooser.showOpenDialog(null);
308
309        if (chooserState == JFileChooser.APPROVE_OPTION)
310        {
311           File file = chooser.getSelectedFile();
312
313           if ((file != null) && (!file.toString().equals(" ")))
314           {
315              addBrowseTxt.setText(file.getAbsolutePath());
316           }
317        }
318     }
319
320     private void itemStateChanged_doWork()
321     {
322        if (comboIndex > 0)
323        {
324           if (addCombo.getSelectedItem() instanceof File)
325           {
326              File file = (File)addCombo.getSelectedItem();
327              addBrowseBtn.setEnabled(true);
328              addBrowseTxt.setText(file.getAbsolutePath());
329           }
330           else
331           {
332              System.out.println("Combo item: " + addCombo.getSelectedItem().toString());
333           }
334        }
335        else
336        {
337           if (comboIndex == 0)
338           {
339              addBrowseBtn.setEnabled(false);
340              addBrowseTxt.setText("All Folders");
341           }
342           else
343           {
344              System.out.println("Combo box selection out of range: " + comboIndex);
345           }
346        }
347     }
348
349     private void showMessage()
350     {
351        /*
352        addCombo.setVisible(false);
353        addLabel2.setVisible(false);
354        addBrowseTxt.setVisible(false);
355        addBrowseBtn.setVisible(false);
356        addLabel3.setVisible(false);
357        addTypesText.setVisible(false);
358        addLabel4.setVisible(false);
359        addLabel5.setVisible(false);
360        addSearchBtn.setVisible(false);
361        addCancelBtn.setVisible(false);
```

```
362         */
363     }
364 }
```

ATTACHMENT 18

```
#include <windows.h>
#include "frontend.h"
#include "winampmgr.h"

int WINAPI DllMain (HINSTANCE hInstance,
                    DWORD fdwReason,
                    PVOID pvReserved)
{
    return TRUE;
}

/*
** int res = SendMessage(hwnd_winamp,WM_WA_IPC,0,IPC_ISPLAYING);
**
** IPC_ISPLAYING returns the status of playback.
** If it returns 1, it is playing. if it returns 3, it is paused,
** if it returns 0, it is not playing.
*/
JNIEXPORT jint JNICALL
Java_WinAmpMgr_getWinAmpStatus(JNIEnv* jnienv, jobject jobj)
{
    HWND hwnd_winamp;
    int res;

    hwnd_winamp = FindWindow("Winamp v1.x",NULL);
    res = SendMessage(hwnd_winamp,WM_WA_IPC,0,IPC_ISPLAYING);
    return res;
}

/*
** COPYDATASTRUCT cds;
** cds.dwData = IPC_PLAYFILE;
** cds.lpData = (void *) "file.mp3";
** cds.cbData = strlen((char *) cds.lpData)+1; // include space for null char
** SendMessage(hwnd_winamp,WM_COPYDATA,(WPARAM)NULL,(LPARAM)&cds);
**
** This will play the file "file.mp3".
**
*/
JNIEXPORT jint JNICALL
Java_WinAmpMgr_playWinAmp(JNIEnv* jnienv, jobject jobj, jstring jstr)
{
    HWND hwnd_winamp;
    COPYDATASTRUCT cds;
    cds.dwData = IPC_PLAYFILE;
    cds.lpData = (void *) jstr;
    cds.cbData = strlen((char *) cds.lpData)+1; // include space for null char
    hwnd_winamp = FindWindow("Winamp v1.x",NULL);
    SendMessage(hwnd_winamp,WM_COPYDATA,(WPARAM)NULL,(LPARAM)&cds);
    return 0;
}

/*
** SendMessage(hwnd_winamp,WM_WA_IPC,0,IPC_DELETE);
**
** You can use IPC_DELETE to clear Winamp's internal playlist.
*/
JNIEXPORT jint JNICALL
Java_WinAmpMgr_clearWinAmpPlayList(JNIEnv* jnienv, jobject jobj)
{
    HWND hwnd_winamp;

    hwnd_winamp = FindWindow("Winamp v1.x",NULL);
    SendMessage(hwnd_winamp,WM_WA_IPC,0,IPC_DELETE);
    return 0;
}

/*
** int res = SendMessage(hwnd_winamp,WM_WA_IPC,mode,IPC_GETOUTPUTTIME);
**
** IPC_GETOUTPUTTIME returns the position in milliseconds of the
** current song (mode = 0), or the song length, in seconds (mode = 1).
** Returns -1 if not playing or error.
*/
JNIEXPORT jint JNICALL
Java_WinAmpMgr_getWinAmpOutputTime(JNIEnv* jnienv, jobject jobj)
{
    HWND hwnd_winamp;
    int res;

    hwnd_winamp = FindWindow("Winamp v1.x",NULL);
    res = SendMessage(hwnd_winamp,WM_WA_IPC,0,IPC_GETOUTPUTTIME);
    return res;
}

JNIEXPORT jint JNICALL
Java_WinAmpMgr_getWinAmpSongLength(JNIEnv* jnienv, jobject jobj)
{
    HWND hwnd_winamp;
    int res;
```

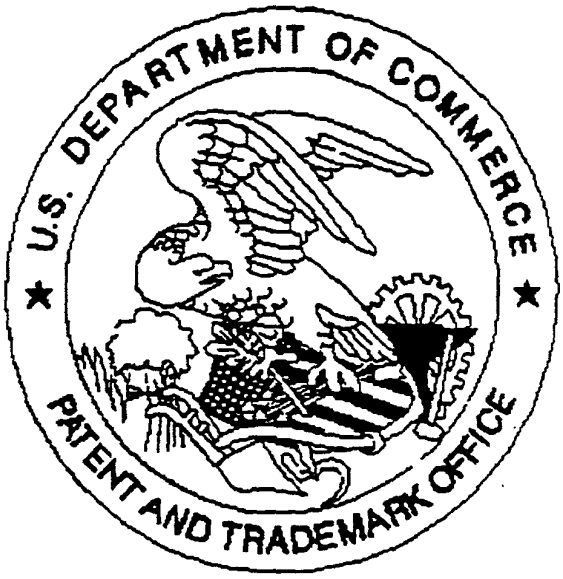
```
91
92          hwnd_winamp = FindWindow("Winamp v1.x",NULL);
93          res = SendMessage(hwnd_winamp,WM_WA_IPC,1,IPC_GETOUTPUTTIME);
94          return res;
95      }
96
97
98      /*
99      ** Sends a message to WinAmp to Quit.
100     */
101     JNIEXPORT jint JNICALL
102     Java_WinAmpMgr_closeWinAmp(JNIEnv* jnienv, jobject jobj)
103     {
104         HWND hwnd_winamp;
105
106         hwnd_winamp = FindWindow("Winamp v1.x",NULL);
107         PostMessage(hwnd_winamp, WM_QUIT, 0, 0);
108         return 0;
109     }
110
111
112     /*
113     ** Sends "button-press" messages to Winamp
114     **   WINAMP_BUTTON2 = Play
115     **   WINAMP_BUTTON3 = Pause
116     **   WINAMP_BUTTON4 = Stop
117     */
118     JNIEXPORT jint JNICALL
119     Java_WinAmpMgr_pressPlayWinAmp(JNIEnv* jnienv, jobject jobj)
120     {
121         HWND hwnd_winamp;
122         int res;
123
124         hwnd_winamp = FindWindow("Winamp v1.x",NULL);
125         res = SendMessage(hwnd_winamp, WM_COMMAND,WINAMP_BUTTON2,0);
126         return res;
127     }
128
129     JNIEXPORT jint JNICALL
130     Java_WinAmpMgr_pressPauseWinAmp(JNIEnv* jnienv, jobject jobj)
131     {
132         HWND hwnd_winamp;
133         int res;
134
135         hwnd_winamp = FindWindow("Winamp v1.x",NULL);
136         res = SendMessage(hwnd_winamp, WM_COMMAND,WINAMP_BUTTON3,0);
137         return res;
138     }
139
140     JNIEXPORT jint JNICALL
141     Java_WinAmpMgr_pressStopWinAmp(JNIEnv* jnienv, jobject jobj)
142     {
143         HWND hwnd_winamp;
144         int res;
145
146         hwnd_winamp = FindWindow("Winamp v1.x",NULL);
147         res = SendMessage(hwnd_winamp, WM_COMMAND,WINAMP_BUTTON4,0);
148         return res;
149     }
150
151     /* (requires Winamp 2.0+)
152     ** SendMessage(hwnd_winamp,WM_WA_IPC,volume,IPC_SETVOLUME);
153     **
154     ** IPC_SETVOLUME sets the volume of Winamp (from 0-255).
155     */
156     JNIEXPORT jint JNICALL
157     Java_WinAmpMgr_setWinAmpVolume(JNIEnv* jnienv, jobject jobj, jint vol)
158     {
159         HWND hwnd_winamp;
160
161         hwnd_winamp = FindWindow("Winamp v1.x",NULL);
162         SendMessage(hwnd_winamp,WM_WA_IPC,vol,IPC_SETVOLUME);
163         return 0;
164     }
165
166
167     #ifdef __cplusplus
168     }
169     #endif
```

ATTACHMENT 19

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include "jni.h"
/* Header for class WinAmpMgr */

#ifndef _Included_WinAmpMgr
#define _Included_WinAmpMgr
#ifdef __cplusplus
extern "C" {
#endif



JNIEXPORT jint JNICALL Java_WinAmpMgr_getWinAmpStatus
  (JNIEnv *, jobject);

JNIEXPORT jint JNICALL Java_WinAmpMgr_playWinAmp
  (JNIEnv *, jobject, jstring);

JNIEXPORT jint JNICALL Java_WinAmpMgr_getWinAmpStatus
  (JNIEnv *, jobject);

JNIEXPORT jint JNICALL Java_WinAmpMgr_clearWinAmpPlayList
  (JNIEnv *, jobject);

JNIEXPORT jint JNICALL Java_WinAmpMgr_getWinAmpOutputTime
  (JNIEnv *, jobject);

JNIEXPORT jint JNICALL Java_WinAmpMgr_getWinAmpSongLength
  (JNIEnv *, jobject);

JNIEXPORT jint JNICALL Java_WinAmpMgr_closeWinAmp
  (JNIEnv *, jobject);

JNIEXPORT jint JNICALL Java_WinAmpMgr_pressPlayWinAmp
  (JNIEnv *, jobject);

JNIEXPORT jint JNICALL Java_WinAmpMgr_pressPauseWinAmp
  (JNIEnv *, jobject);

JNIEXPORT jint JNICALL Java_WinAmpMgr_pressStopWinAmp
  (JNIEnv *, jobject);

JNIEXPORT jint JNICALL Java_WinAmpMgr_setWinAmpVolume
  (JNIEnv *, jobject, jint);



#ifdef __cplusplus
}
#endif
#endif
```

United States Patent & Trademark Office

Office of Initial Patent Examination -- Scanning Division



Application deficiencies found during scanning:

☑ Page(s)_____ of Certificate EXPress mail were not present for scanning. (Document title)

☐ Page(s)_____ of_____ were not present for scanning. (Document title)

☐ ***Scanned copy is best available.***